



Bachelorarbeit

Evaluating the use of Docker containers for Environmental Simulations

Quirin Pamp



Bachelorarbeit

Evaluating the use of Docker containers for Environmental Simulations

Quirin Pamp

Aufgabensteller: Prof. Dr. Dieter Kranzlmüller

Betreuer: Dr Stephan Hachinger
Tobias Weber

Abgabetermin: 3. September 2018

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 3. September 2018

.....
(Unterschrift des Kandidaten)

Abstract

Environmental computing includes a wealth of simulations, often involving special-purpose code and frameworks. Building, configuring and deploying these frameworks to run actual simulations using real world data can involve significant time and effort for anyone not intimately acquainted with the relevant software.

This bachelor thesis examines containerization (via Docker) to reduce the effort involved in the initial configuration and deployment of environmental simulations to a minimum. In addition, the containerization approach is assessed with respect to secondary requirements like reproducibility, transparency, reusability and maintenance. Particular attention is also paid to the interfaces required for input and output data, as well as the run time configuration of containerized simulations.

Throughout the work Flexpart, a particle dispersion model suitable for the simulation of a large range of atmospheric transport processes, as used in the context of the AlpEnDAC and GeRDI projects, is used as an example. A parameterized Flexpart container is created and compared to the exiting virtual machine-based implementation providing on-demand simulations at AlpEnDAC. In the process a more general blue print for the containerization of environmental simulations is provided.

The prototype is tested both on and off the AlpEnDAC infrastructure and evaluated against the given criteria. It is established that both the prototype in particular and the container approach in general offer significant advantages over the virtual machine-based approach. These advantages include increased modularity, a complete absence of overhead, a self documenting and automated build process, a resultant increase in the ease of routine maintenance, and improved ease of use. Collectively these advantages almost certainly justify the relatively modest up-front effort of containerizing environmental simulations.

Future work might involve the containerization of additional simulations, as well as the optimization of docker-based simulations for parallel execution and scalability. Such optimization could involve the use of container-orchestration platforms like Kubernetes.

Contents

1. Introduction	1
2. Background	3
2.1. Containerization and Docker	3
2.1.1. Docker Terminology and Architecture	5
2.1.2. Docker Build Process	7
2.1.3. Runtime Data Interfaces	7
2.2. Environmental Simulations: Flexpart in the context of AlpEnDAC and GeRDI	11
2.2.1. Flexpart	11
2.2.2. AlpEnDAC	13
2.2.3. GeRDI	14
3. Requirements Analysis	17
3.1. Data Interfaces	17
3.2. Runtime Configuration	18
3.3. Ease of Use	19
3.4. Reusability and Maintenance	20
3.5. Reproducibility	21
3.6. Openness, Access, and Freedom	21
3.7. Best Docker Practices	22
4. Prototype Implementation and Blueprint	23
4.1. The Flexpart Prototype	23
4.1.1. Providing the Base Operating Environment	24
4.1.2. Providing the Simulation Framework	25
4.1.3. Providing the Runtime Dependencies	26
4.1.4. Configuring the Simulation	27
4.1.5. Providing the Necessary Data Interfaces	29
4.1.6. Providing Pre- and Post-Processing Tools	30
4.2. Extracting a General Blueprint	32
5. Evaluation and Assessment	35
5.1. Test Setup	35
5.1.1. Functionality Tests	36
5.1.2. Reproducibility Test	37
5.1.3. Virtual Machine Comparison Tests	37
5.2. Evaluation Relative to the Requirements	39
5.2.1. Volumes as Data Interfaces	39
5.2.2. Parameterized Containers for Run-Specific Configuration	40
5.2.3. Ease of Use Requirements	41

Contents

5.2.4. Reusability and Maintenance Requirements	42
5.2.5. Reproducibility Requirements	43
5.2.6. Remaining Requirements	44
6. Conclusion	45
A. List of Git Repositories	49
B. Flexpart Container Quick Start Guide	51
C. Table of User Parameter Tests	53
List of Figures	55
List of Tables	57
List of Listings	59
Bibliography	61
Acknowledgements	65

1. Introduction

The environmental sciences make frequent use of data intensive computational simulations. Such simulations generally involve custom-developed code and frameworks, as well as a wide variety of possible data, stored in a wide variety of possible formats. The compilation and installation, as well as the configuration and deployment of such frameworks to undertake simulations, or even just the acquisition of relevant input data, can represent considerable entry hurdles to those not already intimately acquainted with them. These hurdles are especially high for those involved whose primary background lies within the environmental sciences, and not in software development.

Such entry hurdles that arise when obtaining new (potentially better) data sources, or switching to a different computational framework, can create what Hachinger et al. [WHNW18] call *knowledge and know-how bias*. Simply put, researchers will prefer data sources and computational frameworks they are already familiar with, to others that might have lead to better scientific results.

The problem of how to reduce such entry hurdles (and improve ease of use) is the underlying motivation for the present work. It is evaluated and assessed to what extent containerization using Docker can contribute to a reduction in these hurdles for environmental simulations. This is done in the context of the existing AlpEnDAC and GeRDI projects at the LRZ (“Leibniz Supercomputing Centre”, Garching b.M., Germany). These projects, are broadly aimed at providing research data management and on-demand environmental simulations as a service to scientists.

The AlpEnDAC project (“Alpine Environmental Data Analysis Center”) provides on-demand environmental simulations, as well as the data repositories needed for those simulations [HHM⁺16]. One of the simulations provided uses Flexpart (elsewhere stylized as “FLEX-PART”, for “FLEXible PARTicle dispersion model”), a Lagrangian transport and dispersion model [SFF⁺05], to simulate atmospheric transport processes (e.g., how dust or pollutants are dispersed throughout the atmosphere). The current Flexpart implementation used at AlpEnDAC is virtual machine-based. Since these virtual machines require maintenance unrelated to the needs of the simulations themselves, for example OS updates, and have aspects that are cumbersome to use, there is an interest in switching to a Docker-based alternative. As a matter of methodology, this thesis proceeds by implementing a prototype providing containerized Flexpart simulations for use with AlpEnDAC, which is then used to develop a general approach for the containerization of environmental simulations.

The GeRDI (“Generic Research Data Infrastructure”) project provides a generic reference architecture for research data infrastructures that can be integrated with existing research data management solutions. This architecture is based on self-contained systems with stable interfaces to provide continuous integration for a federated infrastructure [dSHWK18]. It is flexible enough to accommodate the needs of diverse research communities holding highly

1. Introduction

heterogeneous data, all while facilitating data sharing and data driven workflows [GAB⁺17]. The prototype developed for AlpEnDAC is intended also as a basis for an implementation of the reference architecture provided by the GeRDI project. In particular, this adds interface design as an additional important focus of the present work.

The Flexpart prototype in the context of AlpEnDAC and GeRDI is both the first use case, and the principal example of this thesis. However, this thesis aims not only to provide a containerized implementation for Flexpart simulations, but also to help assess the containerization approach for environmental simulations in general. Hence, throughout the process of prototype creation, steps taken are analysed and discussed with respect to their relation to environmental simulations in general. Ultimately, a general blueprint for the containerization of environmental simulations is provided; the details of the Flexpart prototype are of greater interest in so far as they are relevant to this abstraction.

As part of that effort at generalization, the specification of well-defined, stable data interfaces for the prototype in particular and the approach more generally is a central element of this thesis. This will ensure that the particular Flexpart prototype retains the flexibility needed to serve as the basis for a general approach. Such a specification will ensure that the prototype can be easily adjusted to run using local input data, data repositories like those provided by AlpEnDAC, or be extended for full GeRDI integration.

Additional requirements for the prototype concern ease of use, reusability and maintenance, and reproducibility. The prototype is intended to be fully open source to help meet the requirements. Ultimately the prototype is assessed with respect to its stated requirements, as well as to what extent it improves upon these requirements relative to the existing virtual machine-based approach.

Outline of Thesis Chapter 2 starts by providing some background on containerization and docker on the one hand, and the relevant projects relating to environmental simulations on the other. Chapter 3 analyses the requirements for the containerized simulations in detail, guiding the implementation in Chapter 4. Chapter 5 evaluates the containerized simulations with respect to the fulfillment of the requirements, as well as relative to the virtual machine-based virtualization approach. Chapter 6 summarizes the thesis and provides a brief outlook on future work.

2. Background

Following the title of this thesis, this chapter is subdivided into two sections. A section on “Containerization and Docker” (2.1), and a section on “Environmental Simulations” (2.2). These sections provide background information on their respective topics, as relevant to the rest of this thesis. The latter section is approached entirely from the point of view of the concrete projects relevant to this work.

2.1. Containerization and Docker

Containerization refers to the process of running multiple isolated user space instances within a single operating system [Kol06]. Early implementations like `chroot` have been around since at least 1979 [MK79]. More recently, and in particular since the appearance of Docker, containerization has experienced a surge in interest and popularity (see Figure 2.1).

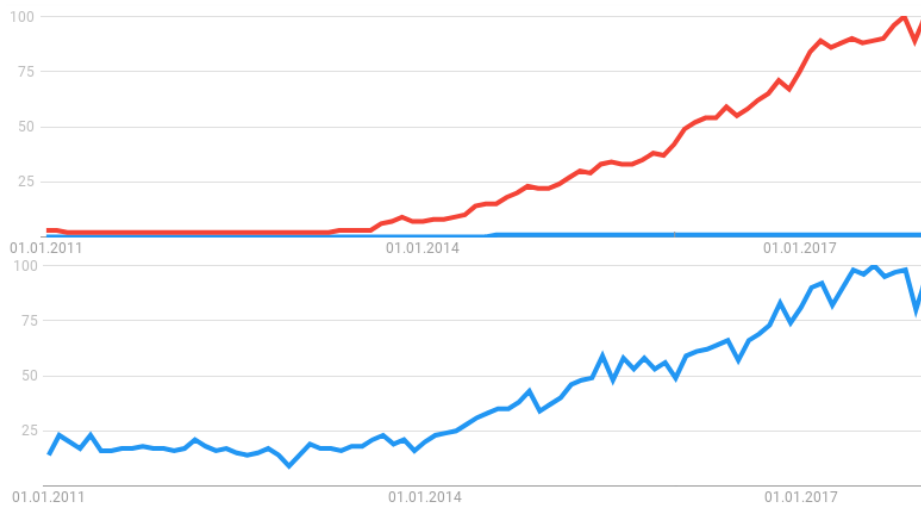


Figure 2.1.: Google Trends for *Docker* (red, scaled to 100 at peak activity over the period provided) and *Operating-system-level virtualization* (blue, repeated with rescaling in the lower pane for readability). As can be seen, searches for Docker practically don’t register prior to Docker’s launch in 2013, at which point the number of searches for both concepts increase significantly, though much faster, to a much higher level for Docker (red vs. blue graph in upper panel).

Data source: Google Trends (<https://www.google.com/trends>).

In comparison to `chroot`, full fledged containerization platforms, like Docker, have expanded the concept. They provide not only encapsulation, but also a single, easy-to-use set of tools

2. Background

that allows for the packaging of software, with all runtime dependencies, for execution in a standardized operating environment [Hog14]. The resultant portability and standardization allows for a large efficiency increase in the delivery of software and services [Kha16]. Once an application or service has been containerized successfully, it should run identically well on any system, using any Linux distribution that has a sufficiently up to date version of Docker installed (without the need for additional testing or configuration).

In addition, Docker Inc. (the company) has invested heavily in the creation of open standards, as well as adherence to the principles of free and open source software [Wal17]. This commitment to transparency is of particular interest to scientific contexts, and has also fostered the growth of a large and vibrant open source ecosystem around Docker and container orchestration.¹

As a result of these advantages, Docker has quickly become an industry standard with respect to containerization in particular, and has also made inroads with respect to virtualization solutions more generally. As a virtualization solution, containers stand in direct contrast to virtual machines, and can be considered a more lightweight alternative. Unlike virtual machines, containers share the operating system kernel with the host system, as well as other containers, removing the overhead of emulating those kernels (see Figure 2.2).

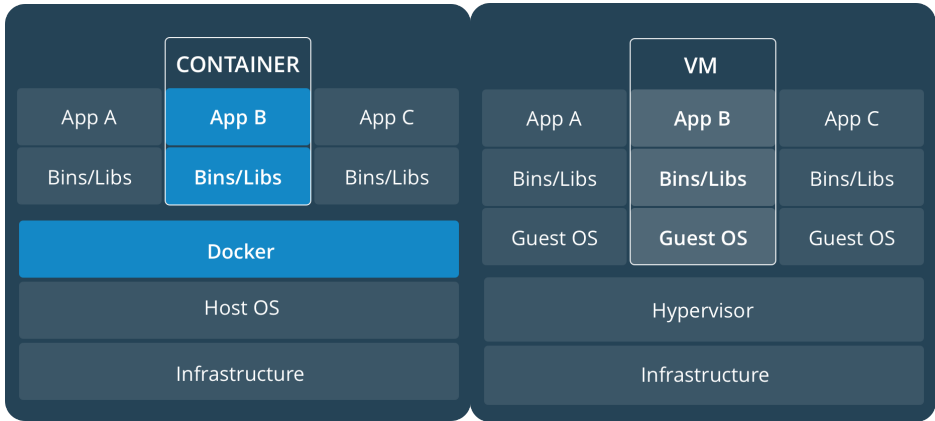


Figure 2.2.: Container vs. virtual machine comparison. Source: Docker documentation²

In addition, the various virtual machine solutions that exist are less consolidated than Docker and the container world. Widespread virtual machine industry standards like VMware have not embraced open source to the same extent.

This thesis is limited to an evaluation of *Docker* containers. The main argument for limiting the scope in this way, is that in terms of adoption, Docker has become the de facto standard for containerization. Nevertheless, it is important to note, that there are alternative contain-

¹ The Docker topic on GitHub includes more than 27000 repositories, related topics like Kubernetes include thousands more. Docker itself is fully open source via the upstream Moby project. See also:

<https://github.com/topics/docker>

<https://github.com/topics/kubernetes>

<https://github.com/moby/moby> (all accessed on July 15th, 2018)

² Containers and virtual machines [Docker v18.03 documentation]. Retrieved July 15th, 2018, from <https://docs.docker.com/get-started/#images-and-containers>

erization platforms like Rkt³ or LXD⁴, that offer largely similar functionality (see [BR17] for a detailed comparison). In fact, these alternatives implement some of the same *actual* container standards, and may share some of the same underlying projects. Rkt claims to use a cleaner process model, with less potential for privilege escalation and may be preferable to Docker from a security point of view.⁵

2.1.1. Docker Terminology and Architecture

In order to understand Docker, care needs to be taken in distinguishing the key concepts involved. Firstly, *images*, *layers*, and *containers* are defined, and the term *encapsulation* is explained:

Images (also referred to as *container images*) are sets of template files, that contain everything needed for Docker to *run a container*.

Layers (also referred to as *intermediary images*) are the constituent parts of container images. Each layer references the layer beneath it via some storage driver. In so doing, each new layer inherits all files and metadata from those below. Each layer corresponds to a directive in the Dockerfile that was used to build the image (see also Section 2.1.2). The concept is visualized in Figure 2.3.

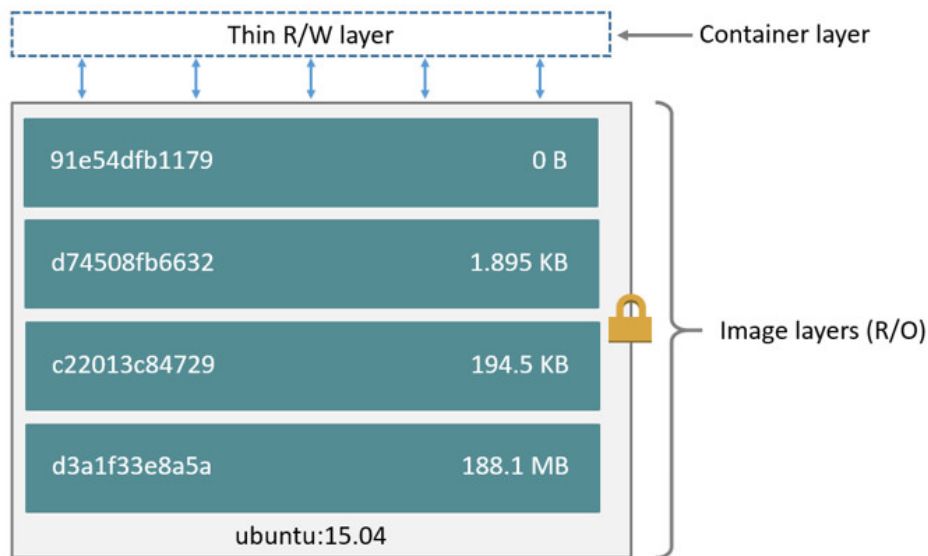


Figure 2.3.: Container and image layers (ubuntu:15.04 image). Source: Docker docs⁶

³ The Rkt website. Retrieved August 25th, 2018, from <https://coreos.com/rkt/>

⁴ The LXD website. Retrieved August 25th, 2018, from <https://linuxcontainers.org/lxd/>

⁵ Rkt vs Docker [rkt 1.29.0 Documentation]. Retrieved July 12, 2018, from <https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html#rkt-vs-docker>

⁶ Images and layers [Docker v18.03 documentation]. Retrieved July 15th, 2018, from <https://docs.docker.com/storage/storagedriver/#images-and-layers>

2. Background

Containers are runnable instances of some *image*. When a container is created, a single read/write *layer* is created, that references the *image* it instantiates (see Figure 2.3). When a container is run, a process is created that can access the relevant read-write layer (and only that layer). As a result, it is possible to create and run (instantiate) many containers on top of a single image.

Encapsulation The process or processes associated with a running container are *encapsulated*. That is, they are isolated from other processes, files not included in the relevant container layer, network traffic not directed at the container, as well as any unneeded system resources of the host system. This isolation relies on various kernel features like cgroups, kernel namespaces, Netfilter, and a union-capable file system.⁷

The Docker platform is made up of several distinct core components, that can collectively be used to manage images and containers. These components and the way they relate to each other describe the Docker architecture (see Figure 2.4).

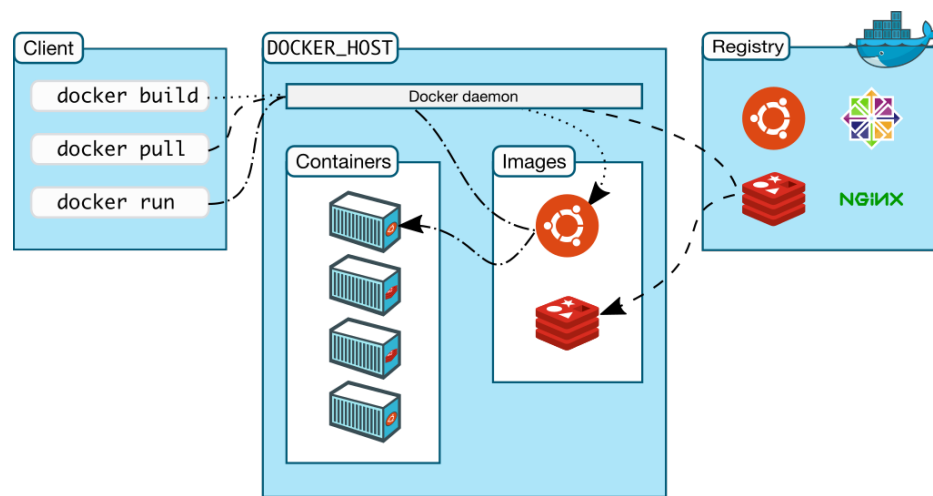


Figure 2.4.: Docker platform architecture. Source: Docker documentation⁸

The Docker daemon is the background program running on a Docker host, that actually manages (creates, deletes, runs, etc.) Docker objects like *images* and *containers*.

The Docker client is the command-line interface that represents the most common way for users to interact with the Docker daemon (the only alternative being direct interaction with the daemon's REST API, possibly via some suitable front end).

The Docker engine refers to the aggregate client-server application, consisting in the Docker daemon, the Docker client, and the daemon's REST API.

Docker registries are used to store images. Images can be pulled from, or pushed to a registry. Docker provides Docker Hub⁹ as a public registry that anyone can use to store and exchange images. Alternatively, private registries can be set up. Without the use of

⁷ The underlying technology [Docker v18.03 documentation]. Retrieved August 4th, 2018, from <https://docs.docker.com/engine/docker-overview/#the-underlying-technology>

⁸ Docker architecture [Docker v18.03 documentation]. Retrieved July 15th, 2018, from <https://docs.docker.com/engine/docker-overview/#docker-architecture>

a registry, images need to be built on, or manually transferred to, each docker host that they are used on.

2.1.2. Docker Build Process

Containerizing an application using Docker, implies building the relevant Docker images. The Docker client provides the `docker build` command (see the Docker build documentation¹⁰) to tell the Docker daemon to build an image from a *Dockerfile* (see the Dockerfile reference¹¹) and a *build context*.

A *Dockerfile* is just a plain text file specifying a list of build instructions that the Docker daemon should execute. A *build context* is a set of files, sent to the Docker daemon, for use in the build process. (For example: Any source code, binaries, library dependencies, or configuration files to be included in the container image.)

The most common scenario simply involves specifying the build context via some path to a folder. Commonly, this folder also contains the relevant Dockerfile. More advanced options, like providing the build context via an archive file, or directly from a Git repository, are also supported.

Once the `docker build` command is invoked, the build context is sent to the daemon, which (after a basic syntax check) proceeds to perform the instructions (also referred to as *directives*) from the Dockerfile. Note that there are directives that can add online content (not supplied by the build context) to the Docker image being built. Each directive in the Dockerfile creates a new layer of the image. If some layer has not changed since a previous build, it is not rebuilt. Images can (and should) be “tagged” with a human readable name and version string.

2.1.3. Runtime Data Interfaces

The previous subsection briefly described the process of creating images. To use such images they are instantiated as a container, which is run as an encapsulated process (see also Section 2.1.1). This encapsulation raises the question of data interfaces. Given a read-only image, how should containers be provided with any additional runtime configuration? How should a simulation running in a container be given access to any input data required? Where should it place the output data generated, for further use?

Since instantiating an image, that is, creating and running a container, involves the creation of an additional read/write layer, the issue of runtime configuration is fairly trivial (it can be added to this final container layer), so long as the containerized application is designed to

⁹ The Docker Hub repository overview website. Retrieved August 25th, 2018, from <https://hub.docker.com/explore/>

¹⁰ Docker build [Docker v18.03 documentation]. Retrieved July 15th, 2018, from <https://docs.docker.com/engine/reference/commandline/build/>

¹¹ Dockerfile reference [Docker v18.03 documentation]. Retrieved July 15th, 2018, from <https://docs.docker.com/engine/reference/builder/>

2. Background

make use of that configuration. Docker provides several ways of providing runtime configuration when a container image is instantiated. Examples include the passing of command-line arguments, setting environmental variables, or mounting configuration files into the container.

For the exchange of larger amounts of data between containers and the outside world, as well as for persistent storage beyond the lifespan of individual containers, there exist two basic approaches:

1. Using various types of filesystem mounts. That is, mounting some filesystem from outside of the container within it.¹²
2. Using a *networked container*. That is, transferring data between the container and the outside world using the networking stack (using arbitrary networking protocols).¹³

The rest of this subsection provides a more detailed look at the individual container data interface mechanisms relevant to this work.

Command-Line Options

Generally all of the mechanisms described in the following subsections need to be triggered by providing the relevant command-line options to the `docker run` command, the command which creates and runs a container/instantiates an image. This command supports extensive command-line options that can be used to override most of the default behaviour defined by the image alone.¹⁴

These runtime options provide the ability to make use of the various types of supported filesystem mounts, as well as the ability to configure the containers' network (for example by providing port forwarding). The `--env`, `-e`, and `--env-file` flags can be used to set environmental variables within the container (see parameterized containers below). Finally, command-line options can also be passed directly to the containerized application.

Parameterized Containers

One of the most common ways to control a container's run time behaviour, is via the use of environmental variables. When invoking the `docker run` command arbitrary environmental variables of the form `ENV_VAR=<value>` may be set using the `--env` or `-e` flag. (For convenience it is also possible to specify a file containing any number of environmental variables using the `--env-file` flag.) These environmental variables will then be visible to the containerized application.

Designing a *parameterized container*, that is, a container that can be configured via parameters stored in environmental variables when it is instantiated, is a matter of designing the containerized application in such a way so as to make use of relevant environmental variables.

¹² Manage data in Docker [Docker v18.03 documentation]. Retrieved July 16th, 2018, from <https://docs.docker.com/storage/>

¹³ Docker container networking [Docker v18.03 documentation]. Retrieved July 16th, 2018, from <https://docs.docker.com/network/>

¹⁴ Docker run [Docker v18.03 documentation]. Retrieved July 16th, 2018, from <https://docs.docker.com/engine/reference/commandline/run/>

Generally via some wrapper script. It is good practice to define any environmental variables within the relevant Dockerfile (using the `ENV` directive) during image creation. This will ensure any needed parameters will always have a default value in case the user does not set them at run time. In addition, it provides a basic indication what environmental variables are supported directly within the relevant container image.

Bind Mounts

A bind mount simply mounts some location from the filesystem of the host machine, to some location within a container. That is, a bind mount is a way for a container to gain direct access to the file system of the host machine. As such, bind mounts weaken the encapsulation of containers, and should not be used in scenarios where the container might be running malicious code.

Use cases for bind mounts include mounting individual configuration files (like `/etc/resolv.conf` for DNS resolution) from the host machine into the container, or sharing source code between the development environment on the host system and a standardized build container.¹⁵

Volume Mounts

Volume mounts are functionally similar to bind mounts, however rather than giving direct access to the file system of the host system, file system access is mediated via a so called *volume driver*. That is, unlike bind mounts, volume mounts are properly encapsulated from the host system by the Docker engine. They can be created using the `docker volume create` command.

This abstraction adds a great deal of flexibility to volumes. The type of file system that is supported is only limited by the availability of a relevant volume driver. For example, it is possible to use a driver that stores data in a remote SSHFS, NFS, or cloud location. Custom volume drivers can be installed as plugins. The volume driver, and by extension the actual file system used, is fully transparent to any containers using the volume.

By default the `local` volume driver is used. This driver simply stores any volume contents under `/var/lib/docker/volumes/<volume_name>` on the docker host. The `/var/lib/docker` folder is managed by the docker daemon.

A single volume can be mounted into multiple docker containers, which allows for interaction between different containerized applications. A volume will also persist beyond the lifetime of any single container that uses it. In summary, volumes are the intended way to handle persistent file system-based data storage within the context of a container infrastructure.¹⁶

¹⁵ Good use cases for bind mounts [Docker v18.03 documentation]. Retrieved July 17th, 2018, from <https://docs.docker.com/storage/#good-use-cases-for-bind-mounts>

¹⁶ Good use cases for volume mounts [Docker v18.03 documentation]. Retrieved August 4th, 2018, from <https://docs.docker.com/storage/#good-use-cases-for-volumes>

2. Background

Networked Containers

The main alternative to mounts is to use the networking capabilities of containers.¹⁷ These principally allow for the use of arbitrary network services as a means of data transfer, even for services for which there isn't any volume driver.

This approach entails all the advantages and disadvantages of networking more generally. Exposing containers on some network may necessitate additional security considerations as well as the required network configuration. The desired network services will generally need to be built into the container image. Remember to check Docker Hub for existing images before attempting to do so yourself. For large distributed infrastructures, networking capabilities are likely to be entirely indispensable. Whether networked containers are desirable, is ultimately entirely dependent on the use case at hand.

Summary of Interface Mechanisms

Table 2.1 briefly summarises the interface mechanisms introduced in Section 2.1.3, along with their principal use cases. For completeness sake, it also includes tmpfs mounts, which were not covered above, since they are not used as part of this thesis. They may be of interest to others using this thesis as a blueprint for containerization.

Interface Mechanism	Use Cases
Command-Line Options	Enabling/using the other interface mechanisms. Passing command line options directly to a containerized application designed to make use of them.
Parameterized Container	Providing simple and easy to use launch-time configuration to a containerized application designed to make use of them.
Bind Mount	Providing host system configuration files to the container. Sharing source code from a development environment on the host system with a running container. Direct access to the file system of the Docker host. Only for non-security sensitive contexts!
Volume Mount	Persistent data storage. Sharing data between multiple containers. Taking advantage of the abstraction offered by volume drivers.
tmpfs Mount	Large amounts of non-persistent state data (for performance). Sensitive data like user passwords, that should not persist.
Networked Container	Anything not covered by the other mechanisms. Taking advantage of existing network services. Providing network services from a container.

Table 2.1.: Summary of runtime and launch-time data interface mechanisms with use cases.

¹⁷ Docker networking overview [Docker v18.03 documentation]. Retrieved August 4th, 2018, from <https://docs.docker.com/network/>

2.2. Environmental Simulations: Flexpart in the context of AlpEnDAC and GeRDI

Unlike “containerization”, the term “environmental simulation” does not admit a single all encompassing (and still illuminating) definition. As a result this section proceeds by describing the concrete simulation and the projects relevant to this work. General background topics relevant to environmental simulations, like *FAIR* research data principles, are touched upon as part of these project descriptions.

2.2.1. Flexpart

Flexpart (elsewhere stylized as FLEXPART, a contraction of “FLEXible PARTicle dispersion model”) is described as a “Lagrangian particle dispersion model that simulates the long-range and mesoscale transport, diffusion, dry and wet deposition, and radioactive decay of tracers released from point, line, or volume sources” [SFF⁺05] and provides the main example for the present work.

In simple terms, Flexpart can be used to simulate the dispersion of atmospheric particles over time. Doing so requires both relevant weather data, more specifically wind data as input, as well as a sufficient *model configuration*. The model configuration includes both a description of the grid, that *particles*¹⁸ are modeled on, as well as a description of all relevant particle and atmospheric properties.

In the context of this work, the primary interest lies in Flexpart from the point of view of software usability and not the details of the underlying physics. Factors relevant to that usability include the initial learning curve, the availability of documentation, the ease of installation and configuration, as well as modifiability, and maintenance. The following subsections offer a brief description of the current state of Flexpart as relevant for these factors.

Note that none of these descriptions are meant to criticise. Instead they are meant to indicate the kind of entry hurdles that may exist, for software that is almost entirely developed by scientists in their own time. Scientists who may themselves be learning best development practices as they go along and as needs arise. These subsections provide anecdotal evidence for the motivation of the present work. On their basis, it can be concluded that Flexpart, in its current state, is very difficult to use without direct person to person help from someone already using it, and that this difficulty is likely to frustrate all but the most determined of potential new users.

Codebase, Versions, and Development Platform

Flexpart is developed in Fortran as free and open source software. Various Fortran standards were used throughout its version history. Flexpart is based on Flextra, a simpler predecessor that has been around since at least 1995 (see [SWSKK95]). As of this writing, the newest

¹⁸ In Flexpart the term *particles* need not refer to actual particles, but can be understood to be analogous to “infinitesimally small air parcels”. See also [SFF⁺05].

2. Background

version of Flexpart has major version number 10. The prototype developed as part of this work makes use of Flexpart version 9 (the newest stable release). Flexpart versions are published as source tarballs, made available for download on the Flexpart website.¹⁹

A development platform comprising of a wiki, a ticketing system, and Git for version control (accessible via the Flexpart website) were introduced in June 2012, as part of the release of version 9.0 of Flexpart, which was released under version 3 of the GNU General Public License. While this platform represents a marked improvement in the availability of resources for Flexpart collaboration, there is little evidence it has encouraged many new contributors to join the effort. Most commits in the Git repository appear to stem from existing core developers.

Overall, the Git repository appears disorganized with unclear branching (workflow) conventions, no apparent naming conventions (for tags, branches, or commit messages), and only sporadic and inconsistent use of same. For example, there are ubiquitous version tags for 9.3.*, but none for earlier versions, as well as some tags that convey no obvious meaning. In other words, the repository appears to suffer from a lack of clear conventions and contribution guidelines.

Documentation and Other Resources

There is a full published technical manual for version 6.2 (published in 2005) which has been updated (and republished) for version 10.0, as well as version 3.1 of Flexpart-WRF (see [SFF⁺05], [PSG⁺17], and [BAS⁺13]). An unpublished version of this manual has been partially kept up-to-date for some additional versions of Flexpart; this version specific documentation is only available with the relevant source tarballs or in the Git repository.

There are some additional resources in the wiki, including basic installation instructions (see the following subsection), as well as the course materials from a Flexpart workshop from 2013 (intended as a tutorial). These resources are generally incomplete and dated. They reference “the Flexpart documentation”, the right version of which is generally difficult to find (see the previous paragraph).

The Flexpart community generally tries its best to reply to any opened tickets, but many of the reported problems relate to system specific difficulties with the installation of library dependencies, and are difficult to diagnose remotely. Such difficulties are generally considered Linux administration issues, rather than Flexpart issues and often end with a recommendation to “talk to your Linux administrator”²⁰.

A “known-to-work” installation environment is one of the things which this thesis aims to deliver via the containerization approach. The result could be integrated into upstream repositories as a contribution to the larger Flexpart community.

¹⁹ See the “Downloads” page of the Flexpart website. Accessed on July 22nd, 2018
<https://www.flexpart.eu/downloads>

²⁰ See ticket 177 (accessed on July 23rd, 2018) for an example:
<https://www.flexpart.eu/ticket/177>

Compilation, Installation, and Configuration

There is a binary package for Debian housing Flexpart 9.02²¹. However, this packaging effort is currently of an insufficient quality for use on production systems (S. Hachinger, private communication), and has not seen much recent activity. The standard way of installing Flexpart then, is to compile it yourself. The kind of research projects involving Flexpart, are sooner or later likely to want to make minor adjustments to code or compile time options anyway.

Flexpart compilation is generally done using the `gfortran` compiler (other Fortran compilers may work, but are untested by the Flexpart developers). Successfully compiling and running Flexpart requires the installation (from source or otherwise) of two library dependencies. These external libraries are needed for the decoding of meteorological input data, as Flexpart uses input data usually available in the GRIB (“GRIdded Binary” or “General Regularly-distributed Information in Binary form”) format.[Wor18]

The recommended GRIB library implementation is GRIB-API (`libgrib-api`).²² The Flexpart developers provide little indication with respect to version level incompatibilities, except to note that a version of GRIB-API $> 1.6.1$ is needed for support of version 2 of the GRIB file format. From trial and error during the initial prototype development, it is known that the newest versions of the library dependencies may not work for various (particularly older) Flexpart versions. The prototype used in this work uses version 1.19.0 of GRIB-API, the newest published version as of this writing is 1.27.0.

The JasPer project (or `libjasper` library) provides JPEG image encoding, as specified in the JPEG-2000 Part-1 standard (ISO/IEC 15444-1) and is a dependency of GRIB-API (rather than Flexpart itself), see [AK00] and [Ada06]. As with GRIB-API, there may be version level incompatibilities that need to be found through trial and error. The prototype uses version 1.900.1 of `libjasper`.

2.2.2. AlpEnDAC

The AlpEnDAC (“Alpine Environmental Data Analysis Center”) project provides research data management, as well as on-demand environmental simulations for high-altitude research facilities within the VAO (“Virtual Alpine Observatory”) project [HHM⁺16]. In particular, this includes fully configured Flexpart simulations (including several user chosen parameters). These simulations are run on pre-configured virtual machines on the 80 node, 640 core IaaS “compute cloud” at the LRZ (“Leibniz Supercomputing Centre”). Simulations can be triggered on-demand via a web-based graphical user interface, which in turn triggers the creation of the needed virtual machines via OpenNebula’s EC2 interface.

In the expectation that it will represent an improvement over the existing virtual machines, the prototype developed as part of this work, is ultimately intended for use with AlpEnDAC.

²¹ Package: `flexpart` (9.02-15), accessed on August 5th, 2018, on <https://packages.debian.org/stretch/flexpart>

²² See the GRIB-API website, retrieved on August 5th, 2018, from <https://confluence.ecmwf.int/display/GRIB/Home>

2. Background

As such, AlpEnDAC’s Flexpart model configuration, as well as its user parameter interface is also used for the prototype.

2.2.3. GeRDI

The GeRDI (“Generic Research Data Infrastructure”) project seeks to further the adoption of “Multidisciplinary and FAIR research data management principles”²³. The acronym FAIR stands for “findable, accessible, inter-operable and reusable” [WDA⁺16]. Understood as a contribution to EOSC (“European Open Science Cloud”), GeRDI is closely entwined with the concepts of *open science* and *open access* [GAB⁺17].

The aim is to “enable all scientists in Germany to store, share, and re-use research data across disciplines”.²⁴ In order to achieve this aim, GeRDI provides a reference architecture for a research data infrastructure based on distinct service domains, implemented as self-contained systems/microservices (see Figure 2.5).

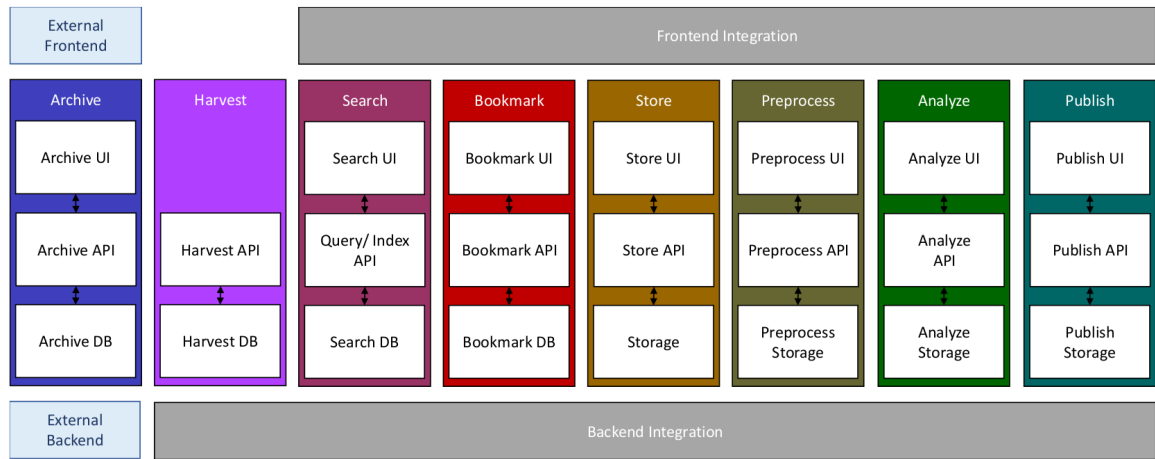


Figure 2.5.: The GeRDI Self-Contained Systems Architecture. [dSHWK18]

In addition GeRDI links this generic architecture, with real life data-driven scientific workflows, by working with so called “Evaluation Communities” to implement its architecture. AlpEnDAC is one such evaluation community. Since the prototype developed as part of this work is intended for use with AlpEnDAC, this raises the question of how it is meant to fit into the larger GeRDI project.

The most important aspect of this connection, is that the containerization approach being developed, can form the basis for implementing GeRDI “Analyze” microservices (see Figure 2.5). When performing an environmental simulation on appropriate input data, “actual analysis on the preprocessed data is performed to gain new scientific insights”, the exact description of “Analyze” from [dSHWK18]. In addition, the containerization approach can

²³ See the GeRDI website, retrieved on August 6th, 2018, from <https://www.gerdi-project.eu/about-gerdi/>

²⁴ See the GeRDI website, retrieved on August 6th, 2018, from <https://www.gerdi-project.eu/about-gerdi/>

easily be extended to provide additional GeRDI microservices in separate containers. The modular nature of services implemented across multiple containers inherently mirrors the modular nature of the GeRDI architecture.

In addition, GeRDI provides some of the philosophical backdrop, context, and motivation for the current work. Both GeRDI and AlpEnDAC view data management on the one hand, and the provision of on-demand simulations on the other, as a service provided to scientists. They do so in a spirit of openness and freedom (in the sense of *free software*). The present work seeks to adhere to that same spirit. Practically this means using open source software and frameworks wherever possible, as well as ensuring every aspect of the prototype is comprehensible and modifiable.

Another aspect of the connection is the fact that GeRDI directly impacts the assumptions surrounding the prototype's input and output data interfaces. That is, the prototype must provide separate well-defined interfaces for input and output data. This leaves open the possibility that input data can be provided to the prototype via a GeRDI implementation, and that output data may be fed back into it. Finally, the prototype's user configuration interface could be connected to a GeRDI implementation, to check if some given simulation has been run before, and need not be run again.

3. Requirements Analysis

This chapter serves to collect all requirements that the finished Flexpart prototype should fulfill. It generally proceeds by introducing some problem or challenge to be solved in each subsection, and then extracting a list of requirements to address said problem or challenge. Both the challenges and the resulting requirements may relate to those in other sections.

3.1. Data Interfaces

Generally speaking, environmental simulations involve large amounts of heterogeneous input and output data. For example, the AlpEnDAC data repository containing the weather data used as input for Flexpart runs, contains more than 22000 files, using some 8.2 TiB of disk space, covering data from 2010 onwards.¹ Individual files cover three hours of data and vary in size from around 15 MiB to over 200 MiB. They are encoded using the GRIB format (see also Section 2.2.1), a standard by the World Meteorological Organization subject to development over time [Wor18]. Flexpart output uses its own binary format and requires custom tools to be read. Other simulation frameworks may use completely different types of data.

As a result of this heterogeneity, the only assumption made about input and output data for the general case, is that it involves files, quite possibly many large ones. Since the chosen approach should serve as a general blueprint, the data interfaces must be kept as general and flexible as possible. Input and output data interfaces should be kept separate, and should support a wide, and extensible variety of local and remote storage options.

These requirements are collected in the following nested list:

- R1) The interface must be able to handle large amounts of data. Ideally, it should only be limited by the underlying hardware, ensuring the software can still be used as technologies and infrastructures continue to improve.
- R2) Output data must be stored in a persistent way.
- R3) Data interfaces should be kept as general and flexible as possible.
 - A) The prototype should have separate interfaces for input data, output data, as well as a separate interface for any needed runtime configuration (see also Section 3.2).
 - B) Native support for local storage, as well as a wide variety of remote storage solutions like NFS, SSHFS, or other cloud-based storage solutions is desirable.

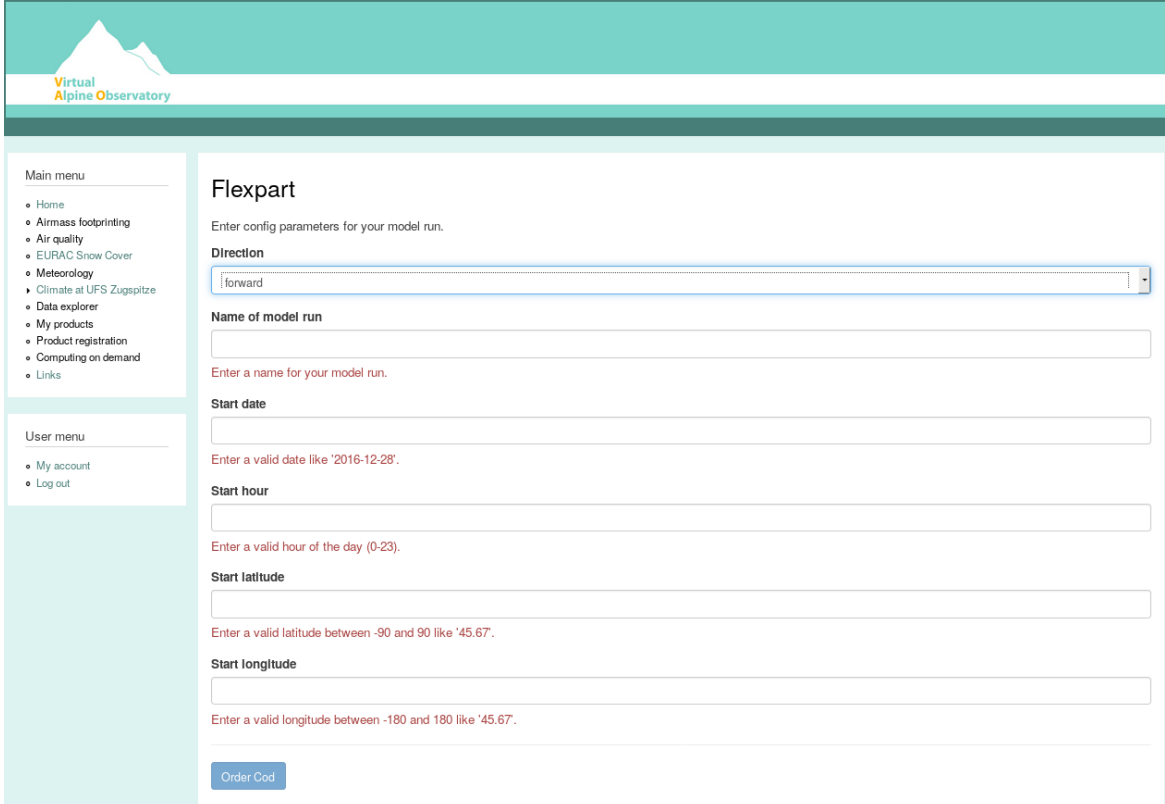
¹ These values were obtained from the AlpEnDAC SSHFS data repository on August 25th 2018.

3. Requirements Analysis

- C) Data should be stored in such a way to allow for a modular extensibility of the containerized application, to perform additional pre-processing or post-processing steps not part of the original workflow.
- R4) The data source on the other side of the interface should be irrelevant to the containerized application (so long as it contains the correct data).
- R5) The interface should remain stable even if the containerized application, or the storage solution is changed.

3.2. Runtime Configuration

As mentioned already (see Section 2.2.2) the present work makes use of the AlpEnDAC model configuration for its Flexpart runs. From all the physical and technical Flexpart input parameters, AlpEnDAC allows the user to set a small subset of run-specific parameters via a web interface (see Figure 3.1).



The screenshot shows the AlpEnDAC web interface for configuring Flexpart runs. The interface has a teal header with the 'Virtual Alpine Observatory' logo. On the left, there is a 'Main menu' with links: Home, Airmass footprinting, Air quality, EURAC Snow Cover, Meteorology, Climate at UFS Zugspitze, Data explorer, My products, Product registration, Computing on demand, and Links. Below this is a 'User menu' with links: My account and Log out. The main content area is titled 'Flexpart' and contains the instruction 'Enter config parameters for your model run.' The form includes several input fields with validation messages: 'Direction' (a dropdown menu showing 'forward'), 'Name of model run' (a text field with the message 'Enter a name for your model run.'), 'Start date' (a text field with the message 'Enter a valid date like '2016-12-28'.'), 'Start hour' (a text field with the message 'Enter a valid hour of the day (0-23).'), 'Start latitude' (a text field with the message 'Enter a valid latitude between -90 and 90 like '45.67'.'), and 'Start longitude' (a text field with the message 'Enter a valid longitude between -180 and 180 like '45.67'.'). At the bottom of the form is a blue button labeled 'Order Cod'.

Figure 3.1.: The AlpEnDAC web interface for the configuration of Flexpart runs.

Source: <https://www.alpendac.eu/> (user account required).

It is a requirement of the Flexpart prototype, that it include a run-time configuration interface, implementing the same parameters that can be set via the AlpEnDAC web interface. The full list of these parameters along with a description, can be seen in Table 3.1

Parameter	Type	Description
Direction	Boolean	Should the simulation run backwards (-1) or forwards (1) in time?
Name/Identifier	String	String to identify the Flexpart run within AlpEnDAC.
Start date	Date string	The start date of the simulation in YYYY-MM-DD format (ISO 8601).
Start hour	Integer	The start hour of the simulation (0-23).
Start latitude	Signed integer	The latitude of the particle source (forward run) or sink (backward run). Must be between -90 and 90.
Start longitude	Signed integer	The longitude of the particle source (forward run) or sink (backward run). Must be between -180 and 180.

Table 3.1.: Runtime configuration parameters for Flexpart runs in AlpEnDAC.

Beyond the specific parameters of the AlpEnDAC web interface, it is paramount that the implementation chosen remains easily modifiable to support arbitrary runtime parameters for arbitrary containerized environmental simulations.

In summary:

- R1) The prototype must implement the AlpEnDAC runtime parameters (see Table 3.1) via a configuration interface.
- R2) The configuration interface must be easily extensible to support arbitrary parameters.

3.3. Ease of Use

As mentioned already (see Chapter 1), the ease of use of the resultant prototype is a major concern. Reducing the entry hurdles for users not acquainted with some simulation framework was the original motivation for this thesis. Existing user communities of simulation frameworks like Flexpart have a significant interest in providing such a reduction, both for the sake of the growth and vibrancy of the community, as well as the ease of access to (and by extension impact of) their own work [HHM⁺16].

A significant part of the entry hurdles associated with large software frameworks, stems from the initial setup. This typically involves compilation, installation and configuration, both of the software itself, as well as any required dependencies. With Docker, most of these steps can be automated. A working Docker installation is presumed throughout this thesis. This is a fair assumption, since well maintained versions of Docker are available through the package manager of every major Linux distribution, and will typically work out of the box. Another important aspect with respect to ease of use is the availability of documentation.

The following list of requirements is extracted from these considerations:

- R1) The number of steps needed to get from a working Docker installation to a first successful test run of the containerized simulation should be kept to a minimum.
- R2) The steps needed to get from a working Docker installation to a first successful test run of the containerized simulation should be fully documented.

3. Requirements Analysis

- R3) Containerized simulations should provide all run-time dependencies as part of the relevant container images.
- R4) The build-time dependencies as well as the build process more generally, should be fully documented.
- R5) Containerized simulations should provide a full model configuration, with appropriate default parameters, ready to run out of the box.
- R6) A relevant subset of the full model configuration should be easily configurable for individual simulation runs via a single, well documented, configuration file.
- R7) The source for any bundled components (like libraries or source code) should be fully documented down to the exact version used.

There is often a trade-off between ease of use on the one hand, and other requirements like modifiability and transparency on the other. However, since Docker containers are anything but a black-box, the effect of such trade-offs can be minimized. Assuming all image sources are provided, users can freely choose to analyse the inner workings of a container to arbitrary depth, or simply use it as is.

3.4. Reusability and Maintenance

Ease of maintenance, as opposed to ease of use, will likely be a major factor for whether the approach outlined in this thesis will find much adoption. If the containerization approach ends up being more maintenance intensive than, for example, existing virtual machine-based approaches, service providers are unlikely to adopt it.

Typical maintenance scenarios include upgrading a containers base operating system, upgrading the containerized application, or modifying an existing solution to accommodate changed workflows or new features. Hence, the relation between maintenance, modifiability and reusability.

The following list of requirements is based on these considerations:

- R1) A containerized simulation must be easier, or at least equally as easy to maintain (upgrade or modify) as the same simulation running from a prepared virtual machine image.
- R2) The present work should provide documentation on how to perform routine upgrades within the containerization approach.
- R3) The approach being developed should seek to be modular in nature whenever possible. This should serve to improve extensibility and modifiability, with minimum impact on existing solutions.
- R4) The prototype should be made fully available via a permissive open-source license using appropriate version control in some appropriate source repository.

3.5. Reproducibility

The reproducibility of results is a basic requirement of all scientific work. At the same time, practical reproducibility in the context of environmental simulations, generally involving large amounts of data as well as large, self developed software frameworks, is not a trivial matter. Minor differences in the version of any software used, including all libraries and dependencies, the choice of compiler and the level of compiler optimization, as well as hardware differences, can all alter the results of complex floating-point computations. This is true even where the model configuration and workflows are well known.

Any reproducibility requirements also intersect with those for ease of use. The more difficult and time consuming it is to reproduce some simulation, the less likely scientists are to do so, even were it might be beneficial.

The resulting reproducibility requirements are formulated as follows:

- R1) Two simulations using identical input data, along with an identical model configuration should yield as near identical output data as possible.²
- R2) Any information needed to reproduce a simulation, including the relevant model configuration, input data source, and the exact version of all relevant software, should be recorded with the output data.
- R3) Given the configuration file of a previously run simulation, as well as access to the relevant input data, rerunning that simulation should be as easy as possible.
- R4) Requirement R4 from Section 3.4 (open-source licence plus version control in some source repository) also applies here.

Note that while R4 above, is not strictly necessary for the reproducibility of individual simulation runs, the full prototype sources, and by extension the full documentation of the build process, is necessary when checking the method of such runs. As mentioned before, the exact compiler used and the level of compiler optimization is part of this method. Since checking the method generally accompanies any effort to reproduce results, R4 should be included as a reproducibility requirement in a broad sense.

3.6. Openness, Access, and Freedom

The notions of openness and freedom, in the sense of “free and open-source software” are a recurring theme throughout this work. The vision of the GeRDI project involves providing open access to research data across disciplines, by building upon the FAIR principles of research data management [MNV⁺17]. AlpEnDAC provides its users with open access to all simulation data, and to measured data as far as authors permit. The present work sees itself in the same spirit of openness.

The main consequences for the present work, are a preference for using open-source software wherever possible, and that the prototype itself be published in this way:

² Complete identity of output data may be impossible for two otherwise identical simulations running on differing hardware.

3. Requirements Analysis

- R1) The prototype should prefer the use of free and open-source software where possible.
- R2) The prototype should itself be published under a permissive open-source license (see also R4 from both Section 3.4 and Section 3.5).
- R3) The prototype should be made available in a public source repository with appropriate version control.

3.7. Best Docker Practices

If the present work is to serve as a blueprint for the containerization of environmental simulations, then it should foster best docker practices from the outset and the following requirement is formulated:

- R1) The prototype should adhere to best Docker practices whenever possible, see also the Docker documentation³.

To fulfill this requirement, the design and implementation chapter (Chapter 4) explicitly discusses any measures taken for the sake of docker “best practices”, even where they are not strictly necessary for the Flexpart example.

³ Best practices for writing Dockerfiles [Docker v18.03 documentation]. Retrieved August 13th, 2018, from https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

4. Prototype Implementation and Blueprint

Conceptually, the problem of how to containerize some given environmental simulation using Docker can be broken down into several distinct subproblems:

1. Providing the base operating environment.
2. Providing the simulation framework itself.
3. Providing the runtime dependencies.
4. Configuring the simulation.
5. Providing the necessary data interfaces.
6. Providing pre- and post-processing tools.

The solution to the individual subproblems can be independently adjusted to the needs of concrete individual simulations like Flexpart. For example, if some given simulation framework is installed from a binary package, no compilation will be needed. Likewise, a given containerized simulation may or may not require pre- and post-processing tools.

The above conceptual division forms the basis for moving from the description of the Flexpart prototype to a generalized blueprint for environmental simulations. That is, it identifies common elements to many disparate environmental simulations.

Section 4.1 describes the development of the AlpEnDAC Flexpart prototype. Section 4.2 abstracts away from the particular Flexpart example to provide a blueprint for environmental simulations in general. Both sections make extensive use of the above conceptual division.

4.1. The Flexpart Prototype

The Flexpart prototype forms the example around which the present work is built. It allows users to run containerized Flexpart simulations using the AlpEnDAC model configuration (with user supplied parameters).

The following subsections provide a detailed description of the prototype and its development. Each subsection is dedicated to a subproblem of the containerization approach, as detailed in the chapter introduction above. Each subsection describes how the individual subproblem was solved for Flexpart, as well as how the chosen solution might relate to other cases.

4.1.1. Providing the Base Operating Environment

While it is possible to create Docker container images “from scratch”, it is far more common to use some “base image” as a starting point. Generally, there are well maintained base images for most major Linux distributions, freely available on Docker hub. Such a “base image” will include a basic Linux directory structure, Bash and other GNU system tools, some core libraries, as well as a locale and environmental variables. In short, everything needed for a basic operating environment. It will usually also include the package management tools of your chosen distribution.

Minimalist base images can be as small as 5 MiB for Alpine Linux¹, or even smaller for Busybox² (which does not provide any package management). Unless there is an overwhelming need for tiny images, it is not generally worthwhile sacrificing the flexibility of a functioning base system and building your images from scratch. In the context of environmental simulations, the size of a base image (around 100 MiB for Debian), is likely to be trivial in comparison to the size of the data involved (see also Section 3.1). As a result, other considerations like ease-of-use and flexibility can be prioritized.

The official Debian repository from Docker Hub³ was chosen as the source for the base container image used for the containerization of Flexpart. Arguments in favor of using Debian include:

- Debian is distributed as free and open-source software, in keeping with the aims of open access and open science (see also Section 3.6).
- Popular Linux distributions like Ubuntu and Mint are based on Debian. As a result, anyone with any previous Linux experience is likely to recognize Debian packaging tools like the `apt-get` command. This is intended to aid the ease of maintenance (see Section 3.4).
- With over 50000 precompiled packages, Debian offers one of the larger repositories of ready to use software and libraries within the Linux ecosystem. See Gonzalez-Barahona et al. [GBRM⁺09] for more on Debian’s scale.
- While Debian’s downstream derivatives like Ubuntu and Mint generally use the same package sources, the Debian release cycle provides simple access both to current and older stable releases, as well as the testing and unstable repositories. This added flexibility is useful in the scientific context, where a mix of newest cutting edge features and specific (older) versions of certain libraries might be needed for some given simulation.
- There is a binary Flexpart package for Debian, providing a choice between installing this package and compiling Flexpart from source.

The above arguments justify the use of a Debian base container for the containerization of Flexpart. However, the choice of base container is at least to some extent a matter of style,

¹ See the alpine repository on Docker Hub. Retrieved August 25th, 2018, from https://hub.docker.com/_/alpine/

² See the busybox repository on Docker Hub. Retrieved August 25th, 2018, from https://hub.docker.com/_/busybox/

³ Debain on Docker Hub. Retrieved August 14th, 2018, from https://hub.docker.com/_/debian/

as well as the specific needs of the project at hand. For example, the image size of around 100 MiB for a Debian container image was not judged to be a major concern. Substituting a different base container if Debian is found to be wanting is a relatively trivial matter.

The command `docker pull debian:stretch` can be used on any Docker host (with a working internet connection) to create a local copy of the relevant Debian base image. Note that the version tag `:stretch`⁴ is specified explicitly, to ensure maximum control over what is pulled from Docker Hub. To use this base image, as part of the prototype, the line `FROM debian:stretch` is included at the top of the Dockerfile.

4.1.2. Providing the Simulation Framework

The software used for some given environmental simulation will often be given as a source repository or tar ball requiring manual compilation. Even if binary packages are available, it may be worth providing the exact sources used in the interests of open access, reproducibility, and modifiability. Compiling software within a docker container advances these goals, by fully documenting the build process via the relevant Dockerfile, as well as by providing a reproducible build environment via the relevant container image.

In the case of the Flexpart prototype, the source files were kindly provided by Dr. Stephan Hachinger. They are the same sources used for the AlpEnDAC project, and provide version 9.0 of Flexpart with some performance optimizations and OpenMP parallelization. The sources are locally stored as part of the build context (see also Section 2.1.2) of the relevant Docker image. The relevant parts of the Dockerfile can be seen in Listing 4.1.

```

1  RUN apt-get update && apt-get install -y \
2      gfortran=4:6.3.0-4 \
3      libgrib-api-dev=1.19.0-1 \
4      build-essential=12.3 \
5      && echo "deb http://deb.debian.org/debian jessie main" \
6          >> /etc/apt/sources.list \
7      && apt-get update && apt-get install -y \
8          libjasper-dev=1.900.1-debian1-2.4+deb8u3
9  WORKDIR /flexpart_code
10 COPY flexpart_code /flexpart_code/
11 RUN make

```

Listing 4.1: Compiling Flexpart

The `RUN` directive (lines 1 and 11) specifies commands that will be run when the container image is built. The environment these commands are run in is given by any image layers already built at that point. The `WORKDIR` directive (line 9) specifies the working directory within the container image (for all further directives). If this directory does not yet exist, it is created. The `COPY` directive (line 10) is used to copy the locally stored Flexpart sources from the build context into the container image. Note that the first `RUN` directive (line 1) installs the build dependencies of Flexpart (via Debian packages), while the second `RUN` directive (line 11) invokes `make` (in the `WORKDIR`), and thus compiles Flexpart.

⁴ “Stretch” is the development code name of Debian 9, and is the currently stable release.

4. Prototype Implementation and Blueprint

The order of these directives, as well as the use of two separate `RUN` directives, are carefully chosen, to maximise the effectiveness of Docker’s use of cache. Recall that every directive in the Dockerfile creates a new image layer. When some directive in the Dockerfile is changed, and the image is rebuilt, unchanged layers can be reused from cache to speed up the process. However, any layers past the first altered layer also need to be rebuilt because of the way layers are based on references to lower layers (see also Section 2.1.2). As a result, it is best practice to place directives that take a long time to execute, or are least likely to change, as early in the Dockerfile as possible. In particular, the above example (Listing 4.1) avoids reinstalling the build dependencies, every time the Flexpart sources are changed.

Note also, how any instances of `apt-get install` are always combined with the `apt-get update` command within a single `RUN` statement. This ensures package lists are always updated before any packages are installed, and is another matter of best Docker practices⁵.

In the above example (Listing 4.1), the `build-essential` and `gfortran` packages provide compilation tools. The `libgrib-api-dev` and `libjasper-dev` packages provide libraries Flexpart is built against (including the needed header files). Lines 5-7 in Listing 4.1 are needed since the `libjasper-dev` package is not available in “Debian Stretch” (but it is available in “Debian Jessie”). In the above example, the exact version of any packages being installed, is always explicitly specified, to meet the documentation requirements laid out in this thesis (see also Section 3.3).

What dependencies and build tools are needed will of course depend on the software that is to be compiled. There are also alternative approaches for the provision of source files to a build container. For example, they might be synchronized directly from a Git repository. If no compilation is needed, this sub problem may simply be solved, by installing the relevant software packages along with any runtime dependencies (see the following subsection).

4.1.3. Providing the Runtime Dependencies

The build time and runtime dependencies of an environmental simulation may differ: In particular, build tools and source code are generally no longer required once a software is compiled. In order to keep the final container image minimal, Docker supports what is known as a *multi-stage build*. Multi-stage builds allow for the specification of two separate container images within a single Dockerfile, while granting the ability to copy files from one of these container images to the next. How this works for Flexpart, is shown in Listing 4.2.

Both the “`builder`” (lines 1-3) and “`runtime`” (lines 5-16) container images use `debian:stretch` as their base container (see also Section 4.1.1). In principle, it would be possible to use two different base containers. For example, one might choose a more lightweight base container for the runtime container image, which is the one that will be used to run simulations. However, the Flexpart case requires some of the same runtime dependencies as buildtime dependencies. Since packages like `libgrib-api-dev` install thousands of new files from a number of dependent packages in various locations, it was considered impractical to copy these files from the buildtime to the runtime container. Instead the same base container is

⁵ Leverage build cache [Docker v18.03 documentation]. Retrieved August 14th, 2018, from https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#leverage-build-cache

used, and the relevant packages are installed along with other buildtime dependencies (lines 7-13 from Listing 4.2).

```

1  #This container is used to compile Flexpart:
2  FROM debian:stretch AS builder
3  #Insert the build steps here (Listing 4.1).
4
5  #This container is used to configure and run Flexpart:
6  FROM debian:stretch AS runtime
7  RUN apt-get update && apt-get install -y \
8      bc=1.06.95-9+b3 \
9      libgrib-api-dev=1.19.0-1 \
10 && echo "deb http://deb.debian.org/debian jessie main" \
11     >> /etc/apt/sources.list \
12 && apt-get update && apt-get install -y \
13     libjasper-dev=1.900.1-debian1-2.4+deb8u3 \
14 && rm -rf /var/lib/apt/lists/*
15 WORKDIR /flexpart_model
16 COPY --from=builder /flexpart_code/flexpart.gfs /flexpart_model/

```

Listing 4.2: The Flexpart multi-stage build

The (runtime) dependencies in the Flexpart example include `libgrib-api-dev` (line 9) and `libjasper-dev` (line 10-13) in the exact same version used in the `builder` container. They do not include the `build-essential` or the `gfortran` packages from the `builder` container (see Listing 4.1). Finally, the `bc` package (line 8) is included. This package is actually a dependency of the configuration scripts used to implement the runtime configuration interface of the Flexpart container (see also Section 4.1.4), not of Flexpart itself.

The `rm` command in line 14 of Listing 4.2, removes around 100 MiB of package lists (downloaded by the `apt-get update` command) from the final container image. This simply serves to minimize the image size at no additional cost. Specifically, the `rm` command is part of the existing `RUN` directive and does not create an additional layer.

Finally, the `COPY` directive (line 16) using the `--from=builder` option, transfers the finished Flexpart binary (`flexpart.gfs`) from the `builder` to the `WORKDIR` of the `runtime` container image.

4.1.4. Configuring the Simulation

Generally speaking, an environmental simulation framework (like Flexpart) can be configured to simulate a large range of different use cases. For example, Flexpart allows for the specification of everything from the composition of the atmosphere, to the particle emissions being simulated, and the geography of the grid. Many of these parameters are likely to be kept fixed for a large number of simulations, while some will be changed for each run. If the dispersion of a particular pollutant is simulated, the geographic location, the amount of the emission, as well as the start and end date, might be varied, while the geography and atmospheric composition remain constant.

4. Prototype Implementation and Blueprint

As a result, the problem of configuring the simulation can be further subdivided as follows:

1. Providing a basic model configuration for some example simulation.
2. Identifying any parameters that need to be modified for each run.
3. Providing a configuration interface for the identified parameters.

Steps 1. and 2. above are highly specific to the concrete simulation being containerized, as well as to the reason for which it is being containerized. Different projects might require more, or less variable parameters even when dealing with the same simulation framework. As a result the approach chosen for step 3. should remain easily extensible (see also Section 3.2). It will be the main focus of this section.

For the Flexpart example, the basic model configuration already in use for AlpEnDAC simulations (graciously provided by Dr. Stephan Hachinger) is reused. It is statically included in the relevant container image. The parameters that need to be modifiable at runtime are those that are provided by the AlpEnDAC web interface (see also Section 3.2).

```
1  #This container is used to compile Flexpart:
2  FROM debian:stretch AS builder
3  #Insert the compilation steps here (lines 1-10 from Listing 4.1).
4  COPY flexpart_model /flexpart_model/
5
6  #This container is used to configure and run Flexpart:
7  FROM debian:stretch AS runtime
8  #Insert the runtime dependency steps here (lines 7-14 from Listing 4.2).
9  WORKDIR /flexpart_model
10 COPY --from=builder /flexpart_model /flexpart_code/flexpart.gfs \
11     /flexpart_model/
12 ENV PATH="${PATH}:/flexpart_model" \
13     INPUT_DATA_MOUNT_POINT="/flexpart_input" \
14     OUTPUT_DATA_MOUNT_POINT="/flexpart_output" \
15     DIRECTION="-1" \
16     RUN_NAME="flexpart_run" \
17     START_DATE="2016-09-19" \
18     START_HOUR="00" \
19     LATITUDE="47.5" \
20     LONGITUDE="11"
21 ENTRYPOINT ["flexpart.sh"]
22 CMD ["flexpart.gfs"]
```

Listing 4.3: The Flexpart model configuration and parameterization

The basic model configuration, is just a set of plain text configuration files that are copied from the build context, into the `builder` container (line 4 from Listing 4.3). From there it is copied into the `runtime` container (line 10) along with the Flexpart binary. It could also be copied directly into the `runtime` container, but this would necessitate a separate `COPY` directive, and hence an additional layer in the final container image.

The `ENV` directive (lines 12-20) is used to define default values for all parameters needed for

the runtime configuration of the containerised simulation (see also Section 3.2). At runtime, these parameters will be available to the containerized application as environmental variables, and can be freely overwritten by the user invoking the `docker run` command.

In order to make use of these environmental variables, that is, in order to implement the runtime configuration interface within the container, the `ENTRYPOINT` and `CMD` directives are used (lines 21-22). When both directives are specified as in the above example, the argument of the `CMD` directive (`flexpart.gfs`) is passed to the argument of the `ENTRYPOINT` directive (`flexpart.sh`) as a console argument, when the container is run.

In other words, `flexpart.sh` becomes a wrapper script for the execution of the Flexpart binary. `flexpart.sh`, along with other scripts called from within it, simply resides within the `flexpart_model` folder of the build context, and is copied into the container along with it. This bash script calls on a separate configuration script, for each Flexpart configuration file that needs to be modified using a parameter from the environmental variables. These configuration scripts perform basic sanity checks on the user supplied parameter within relevant environmental variables, and then overwrite the relevant configuration file using any changed parameters. If a sanity check is failed, a meaningful error message is printed to `stderr` and the Flexpart run is aborted. Otherwise, once all parameters have been processed, the Flexpart binary is executed.

Extending this configuration interface with additional parameters, is simply a matter of adding the relevant environmental variable to the `ENV` directive, and modifying or adding the relevant configuration script. It will require additional sanity checks for the new parameter, as well as the parameters placement within the configuration file contained within it. Afterwards the container image will need to be rebuilt.

Note that the `ENV` directive in Listing 4.3 contains several environmental variables not required by the AlpEnDAC web interface (see Section 3.2). Firstly, the `flexpart_model` folder, containing the Flexpart binary, is added to the container’s `PATH` variable. The `INPUT_DATA_MOUNT_POINT` and `OUTPUT_DATA_MOUNT_POINT` variables are used by the data interfaces discussed in the following section (4.1.5).

4.1.5. Providing the Necessary Data Interfaces

As mentioned in the background chapter (see also Section 2.1.3) there are two basic approaches for the transfer of large amounts of data into or out of a Docker container. These are volumes on the one hand, and networked Docker containers on the other. Volumes naturally fulfill the requirements that were defined for the needed data interfaces (see also Section 3.1). In addition, volumes are easier to use “out of the box” when compared to networked containers. For a networked container, the necessary network services need to be built into the container and may require extensive configuration. These added features will require maintenance and increase the overall container size. They also reduce a container’s modularity since it now contains not only the containerized application, but also the relevant network services.

Finally, it is possible to combine the volume approach with a networked container where necessary. Once data has been transferred from the main containerized application to some

4. *Prototype Implementation and Blueprint*

volume, a separate networked container could access that same volume to provide any needed network services.

For all of the reasons stated above, the Flexpart prototype developed as part of this work, relies entirely on volumes for its input and output data interfaces. Volumes are shown to work, both for data stored locally on the Docker host, as well as with direct access to AlpEnDAC’s NFS- and SSHFS-based data repositories.

From the point of view of the containerized simulation using volumes, the stable data interfaces for input and output data are really just mount points. The Flexpart prototype expects the needed input data to be present in a specific folder within the container and will write the output data to another well-defined folder (`/flexpart_input` and `/flexpart_output` respectively).

This means that, a volume containing the relevant input data, as well as one for the output data need to be created before the container can be run. The AlpEnDAC data repository for Flexpart input data is available via both NFS and SSHFS. And both NFS and SSHFS mounts can be made directly available in a volume via plugins providing the needed volume drivers.

The SSHFS docker volume plugin “`docker-volume-sshfs`” is available on Github⁶, and can simply be installed on an existing Docker host, by issuing the command `docker plugin install vieux/sshfs`. Once installed, a volume can be created using the usual information needed for SSHFS mounts. When the Flexpart simulation is run, it will need to use the `--mount source=<input_volume_name>,target=/flexpart_input` option, and the containerized application will have access to the input data stored in the SSHFS location.

The “`docker-volume-netshare`” plugin⁷ provides similar functionality for NFS-based locations. In general, any remote storage solution for which a pre-existing volume driver can be found, can be used for any containerized application using volumes as a data Interface.

Where no volume driver is available, there are three possible solutions: Change the storage solution to one supported by a volume driver, create a new volume driver, or create an interface to interface container, that will connect to the storage solution and transfer any needed input data into a local volume. None of these approaches are prohibitive.

4.1.6. **Providing Pre- and Post-Processing Tools**

Some simulations may require pre-processing steps on input data provided before the simulation can be run. Others may require additional post-processing tools to make use of the output data. The input data repositories provided by AlpEnDAC are already fully configured for use with Flexpart, but the Flexpart example falls squarely in the latter category. As mentioned before (see Section 3.1), Flexpart output is mostly stored in binary files with a custom format. The Flexpart website lists various libraries and tools, that can be used to

⁶ Docker volume plugin for SSHFS. Retrieved August 19th, 2018, from <https://github.com/vieux/docker-volume-sshfs>

⁷ Docker NFS, AWS EFS & Samba/CIFS Volume Plugin. Retrieved August 19th, 2018, from <https://github.com/ContainX/docker-volume-netshare>

process these output files⁸.

The QuickLook tool, developed by Radek Hofman⁹, is a simple command line-based python script, used to plot and visualize Flexpart output. Both as a proof of concept, and to meet AlpEnDAC’s practical requirements, a version of this tool was containerized in its own separate Docker container. It is a matter of best Docker practices (see also Section 3.7) that each container should provide exactly one application (along with any dependencies and needed configuration). These separate containers can then interact to provide a consistent service. This inherently modular approach, holds obvious benefits for the maintenance and extensibility of container-based services. Individual containers can be freely altered, added, or removed with minimal impact on the rest of the service. In addition, the approach ensures individual Dockerfiles remain simple and comprehensible.

Some of the same steps described in the previous subsections to provide the main Flexpart container, are reused to design the QuickLook container image. `debian:jessie` is used as a base image. This older version of Debian (oldstable) was used, since the `quick_look.py` version provided (v0.2), is from February 2015 and requires older dependencies. Since python is an interpreted language no compilation, and hence no multi-stage-build is required. The QuickLook runtime dependencies are shown in Listing 4.4.

```

1  #Install quicklook-gif dependencies:
2  RUN apt-get update && apt-get install -y \
3      python=2.7.9-1 \
4      python-numpy=1:1.8.2-2 \
5      python-matplotlib=1.4.2-3.1 \
6      python-mpltoolkits.basemap=1.0.7+dfsg-1 \
7      imagemagick=8:6.8.9.9-5+*
```

Listing 4.4: The QuickLook runtime dependencies in the Dockerfile

The QuickLook sources are included directly in the container’s build context via a fork of the upstream Git repository (as a Git submodule). See also Appendix A. That way, multiple versions of QuickLook can be containerized, by pointing to different places in the submodule from different points in the tree of the parent Git repository.

For the configuration interface the QuickLook container also uses the same basic approach as the main Flexpart container: It is parameterized using environmental variables, that are processed by a wrapper-script within the container. The main `quick_look.py` script, is itself intended as a command-line tool providing more than 20 command-line options. Some of these options, in particular the name of the Flexpart output data folder, that is to be operated upon, will need to be changed with almost every QuickLook run. To save the hassle of needing to alter a configuration file for every QuickLook run, a select number of QuickLook options can be passed directly to the script via the `docker run` command. This will override any corresponding parameter supplied as an environmental variable. For example, the command `docker run flexpart/quicklook -h` will print a help message describing possible

⁸ FLEXPART postprocessing tools. Retrieved August 20th, 2018, from <https://www.flexpart.eu/wiki/FpOutput>

⁹ The QuickLook Git repository. Retrieved August 20th, 2018, from <https://bitbucket.org/radekhofman/quicklook>

4. Prototype Implementation and Blueprint

command-line options from within the container.

Finally, the QuickLook container requires a Flexpart output data volume, previously filled with data by a Flexpart run, to be mounted at `/flexpart_output`. (Use the `--mount source=<flexpart_output_volume>,target=/flexpart_output` with the `docker run` command.

As a final (full) example, the command `docker run --mount source=<flexpart_output_volume>,target=/flexpart_output flexpart/quicklook -o <flexpart_output_folder>` will create a GIF format Flexpart visualization as seen in Figure 4.1. This example visualization uses the default configuration of the QuickLook container.

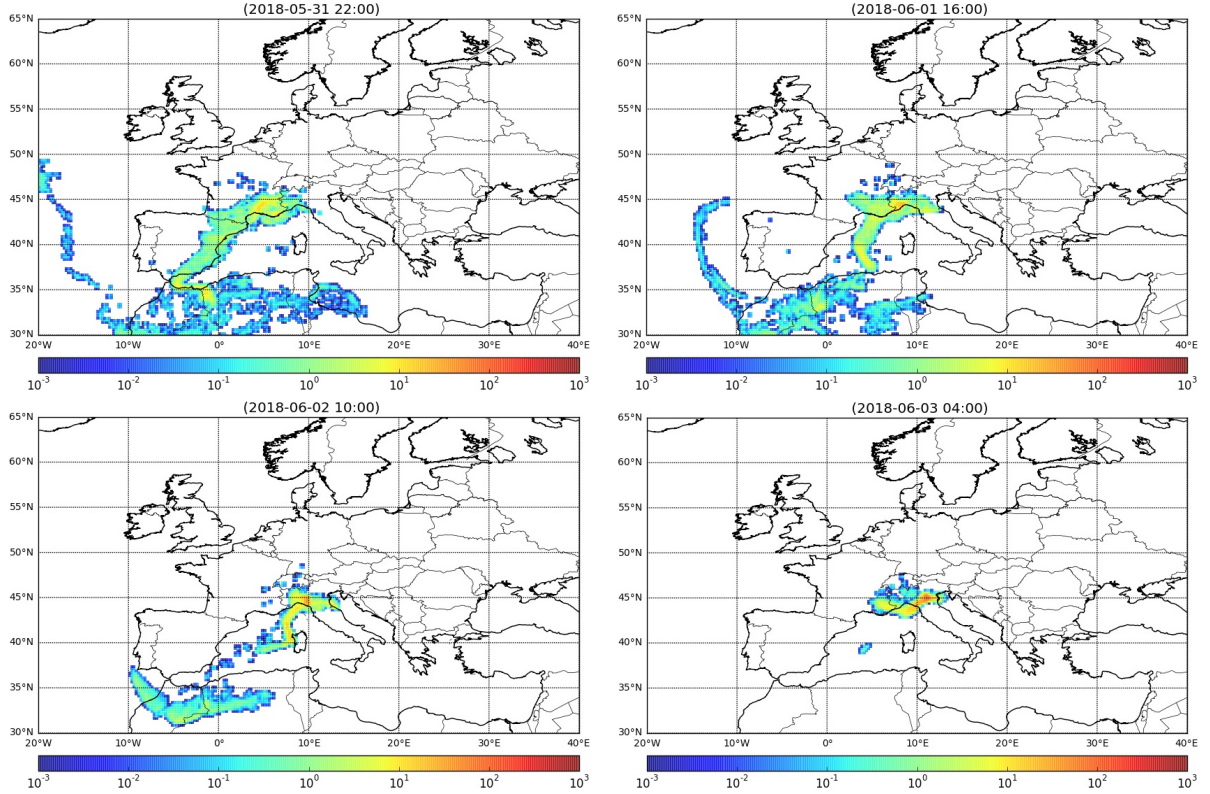


Figure 4.1.: Quicklook container visualization example using default configuration. The Figure shows frames 00, 18, 36, and 45 of a 72 frame GIF. The data is from a backwards Flexpart run.

4.2. Extracting a General Blueprint

The previous section (Section 4.1) provided a detailed description of the containerized Flexpart application prototype that is the main example of this work. Extrapolating from this concrete implementation, a general blueprint for the containerization of environmental simulations can be provided. The basic architecture of this general approach, is given in Figure 4.2.

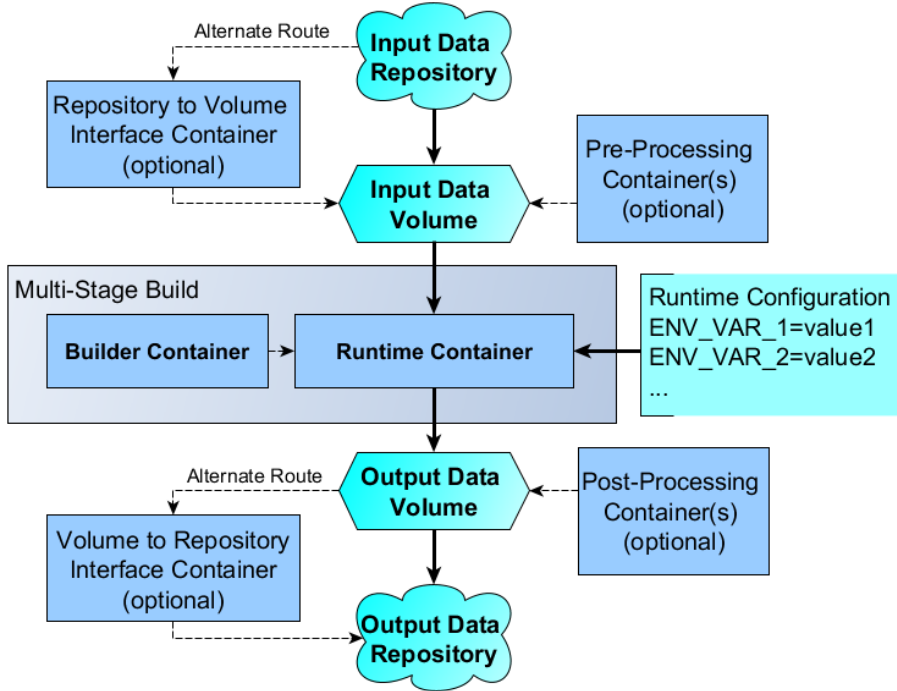


Figure 4.2.: An architecture for the containerization of environmental simulations

The multi-stage build for the main simulation container is explicitly included, in the expectation that most environmental simulation frameworks will require compilation. Even where a precompiled binary installation is available, a fully documented build process as provided by the *builder container* in a multi-stage Docker build, is preferable from an open-access point of view.

The main data flow through the containerized application runs through the central axis of Figure 4.2. Input data is stored in some (possibly cloud-based) *input data repository*, that can ideally be accessed directly through a Docker volume using an appropriate volume driver. If no such volume driver is available, or if a local data buffer is desirable for other reasons, an additional *repository to volume interface container* could be provided that copies needed input data into a local Docker volume, providing an *alternate input data route*. If additional pre-processing of the input data is required, this can now be provided by dedicated *pre-processing containers*.

At this point the main simulations *runtime container* can be started along with any additional run specific configuration parameters (see the *runtime configuration* box on the right of Figure 4.2). The output data is written to an *output data volume*, and the output data workflow (bottom half of the figure) mirrors the one for the input data.

There are many possible extensions to this basic architecture. If a single input, output, or configuration interface proves insufficient for some given simulation, additional interfaces could be defined. Some workflows might require the transfer of entire configuration files into the main runtime container (as an example). For simulations that can be efficiently

4. Prototype Implementation and Blueprint

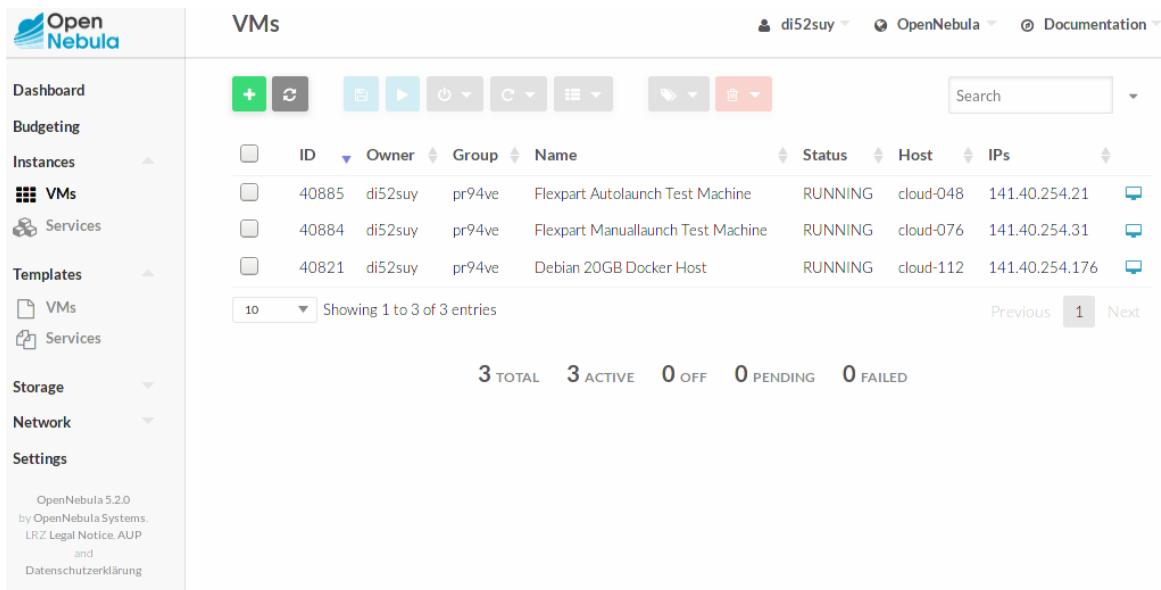
parallelized the single runtime container from Figure 4.2 could be replaced by any number of containers, possibly instantiated from a single image, working together on the same data volumes.

5. Evaluation and Assessment

The focus of this chapter is the evaluation of the Flexpart prototype relative to the assessment criteria and requirements laid out in Chapter 3. Part of this evaluation involves functionality and performance tests, both on and off the existing AlpEnDAC infrastructure. The first purpose of these tests is to verify that the prototype does in fact work as expected, as well as to ensure that there are no major unforeseen performance issues with the concrete implementation. Moreover, the tests allow comparison with the functionality of the existing virtual machine-based approach.

5.1. Test Setup

The test setup involves several distinct components for the various tests performed. All infrastructure-based tests are run in one of three types of pre-configured virtual machines on the LRZ compute cloud (see also Section 2.2.2). This cloud allows users to manage and run their own virtual machines via an Open Nebula web interface (see Figure 5.1). Once created, virtual machines can be accessed via SSH with full root access.



ID	Owner	Group	Name	Status	Host	IPs
40885	di52suy	pr94ve	Flexpart Autolaunch Test Machine	RUNNING	cloud-048	141.40.254.21
40884	di52suy	pr94ve	Flexpart Manuallaunch Test Machine	RUNNING	cloud-076	141.40.254.31
40821	di52suy	pr94ve	Debian 20GB Docker Host	RUNNING	cloud-112	141.40.254.176

3 TOTAL 3 ACTIVE 0 OFF 0 PENDING 0 FAILED

Figure 5.1.: The Open Nebula web interface showing the three types of virtual machines used for tests

The “Debian 20GB Docker Host” virtual machine from Figure 5.1 is the test system for the containerized Flexpart prototype developed as part of this work. It was instantiated

5. Evaluation and Assessment

from a virtual machine template containing Debian GNU/Linux 8.11 (Jessie). Once instantiated, version 18.06.0-ce (build 0ffa825) of Docker was installed using the instructions from the Docker documentation¹. The `flexpart/alpendac-prototype` and `flexpart/quicklook` image sources were obtained via a `git clone` from the `Quba42/flexpart_containerization` repository (see also Appendix A) and built locally on the virtual machine. Many of the tests were run using container images built from the `testing` or `testing-generic` branches, which use a fixed test configuration for comparability.

The other two virtual machines from Figure 5.1 are mock-ups of the virtual machines used in the existing AlpEnDAC implementation. The relevant virtual machine templates were kindly provided by Dr. Stephan Hachinger. The “Manuallaunch” version of this virtual machine is intended for manual instantiation via the Open Nebula web interface. The pre-configured Flexpart run contained within it, can then be launched from within the virtual machine using SSH.

The instantiation of the “Autolaunch” version is intended to be triggered through the EC2 interface. Triggering is achieved by running a PHP script on an Apache server. Crucially this PHP script allows for user data to be injected into the virtual machine being instantiated. If this injection data starts with `#!` it will be recognized as a script, which will be run once the virtual machine has booted. This mechanism forms the basis for the automatic launch of Flexpart runs using user supplied parameters in the current AlpEnDAC implementation.

All three types of test virtual machine were always instantiated using the same compute profile consisting in 32GB of RAM, 4 CPUs, and 8 VCPUs.

All local tests were run on the same laptop used to develop the prototype. The system was Gentoo Linux with kernel version 4.12.12, and Docker version 18.03.1. The CPU model is a Intel(R) Core(TM) i5-4200H with 2.80GHz (x86_64). The main purpose of the local tests was to check the prototypes functionality, as well as to check for reproducibility of Flexpart runs on a different system.

5.1.1. Functionality Tests

The functionality tests consisted in repeatedly varying the values of the environmental variables in the run specific configuration file, and checking whether the Flexpart run completed successfully with sensible output data as visualized using QuickLook. A list of run specific configurations that were tested can be seen in Appendix C.

Note that the functionality tests were run on the `master` branch of the `flexpart_containerization` repository, rather than one of the `testing` branches, where the configuration interface is purposefully fixated. The whole point of the basic functionality tests, was to test the real world functionality of the actual prototype and not of some test version.

Every possible environmental variable from the configuration interface (see also Section 3.2) was varied at least once, thus providing basic test coverage to indicate that a working implementation of the AlpEnDAC runtime parameters has been achieved (see also requirement R1 from Section 3.2).

¹ Get Docker CE for Debian [Docker v18.03 documentation]. Retrieved August 26th, 2018, from <https://docs.docker.com/install/linux/docker-ce/debian/>

In addition to testing the configuration parameters, the volume input interface was tested both with input data stored locally on the Docker host, and with input data stored in both the NFS- and SSHFS-based AlpEnDAC repositories. To this end the `docker-volume-sshfs` and the `docker-volume-netshare` volume drivers were installed on the Docker test hosts (see Appendix A).

5.1.2. Reproducibility Test

The reproducibility test involves running a simulation using identical input data and an identical model configuration (with identical user chosen parameters) both on the LRZ compute cloud, and as a local test, and comparing the output data.

For the initial run, where the `march=native` flag was used for the compilation of Flexpart there was some divergence between the output of the local and the cloud-based runs (see Figure 5.2). Note that the `march=native` flag defaults to `march=haswell` on the local test laptop, and to `march=x86_64` on the virtual machines.

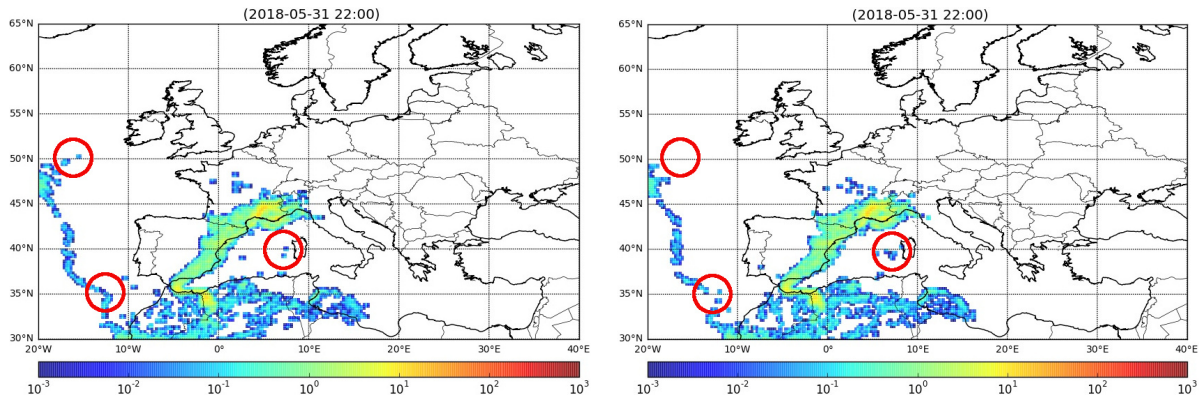


Figure 5.2.: QuickLook visualization of the `march=native` reproducibility test. The left hand pane shows the output from the local run, the right hand pane that from the virtual machine. Three example differences are highlighted using red circles.

However, by explicitly specifying the `march=core2` flag on both the test laptop and the cloud-based virtual machines and rebuilding the Flexpart image, the differences in the output data could be eliminated entirely. The `diff` command, run over the two output folders showed no differences in the resultant binary files.

This result illustrates that there is indeed a trade-off between increased compiler optimization and reproducibility, and that the `march=native` flag should be avoided from a reproducibility point of view.

5.1.3. Virtual Machine Comparison Tests

Though pure run time performance is not a major focus of this work, it is important to establish there are no major unforeseen problems with the prototype. To this end, the time

5. Evaluation and Assessment

to run the basic four day Flexpart simulation used at AlpEnDAC, using a fixed configuration, including fixed input data, was measured both on and off the AlpEnDAC infrastructure using both the “Debian 20GB Docker Host” virtual machine, as well as the “Flexpart Manual-launch Test Machine” (containing the existing AlpEnDAC implementation). In each of these cases input data stored locally on the relevant host was used to avoid any undue influences from networking effects. Since it is not possible to control for capacity fluctuations in the underlying cloud infrastructure, repeated measurements were taken.

Running ten repeats of this simulation on the local test laptop, yielded a value of 380 ± 10 s (6 minutes and 20 seconds) for the chosen four day Flexpart run. The quoted error is the standard deviation of the sample. The same test run using the containerization prototype compiled from the same test branch (`testing-generic`) on the “Debian 20GB Docker Host” virtual machine in the LRZ cloud (also with 10 repeats) yielded a value of 392 ± 8 s (6 minutes and 32 seconds). By comparison the version of Flexpart contained in the “Flexpart Manuallaunch Test Machine”, that is the version of Flexpart used in the current AlpEnDAC implementation, took 844 ± 4 s (or 14 minutes and 4 seconds) for the same run.

This discrepancy is unlikely to be related to any differences between virtual machines and Docker. The infrastructure-based Docker test run was after all itself run within a virtual machine. It does allow us to conclude that, as a matter of contingent fact, the actual prototype implementation performs no worse than the version of Flexpart currently in use (in fact it appears to be running significantly faster at first glance). A detailed analysis of Flexpart performance optimisation, including where the above discrepancy might stem from, lies outside the scope of this work. For readers wishing to look into this further, the following paper might provide a useful starting point. [MK12]

For the final and perhaps most pertinent measurement, the PHP injection mechanism of the “Flexpart Autolaunch Test Machine” was used to measure the launch time of a virtual machine in the LRZ cloud. That is, the instantiation of the relevant test virtual machine, was triggered remotely via PHP script, just as it would be in the current production system at AlpEnDAC. However, instead of injecting a data product ID for the retrieval of the user parameters, a simple `date` command was injected. The triggering PHP script also recorded the time at which instantiation was requested. The difference between these two times results in the time taken to instantiate and boot the virtual machine to the point where a user injected script is first executed.

This measurement is so interesting, since it represents an inherent overhead of the virtual machine-based approach, that can be all but eliminated using containers. Note that instantiating a docker container has no measurable overhead. [FFRR15] It simply does not take significantly longer to launch an encapsulated container based process on a Docker host, than it would to launch any other process. The value for the virtual machine uptime was measured to be 80 ± 10 s using 20 repeated measurements. For Flexpart runs taking in the order of tens of minutes, that extra minute and 20 seconds is significant. This is especially true, if we want to build modular extensions to the original workflow, necessitating separate containers. This would not be feasible for the virtual machine-based approach. A detailed performance comparison of different virtual machine and container systems, going far beyond the uptime measurement supplied here, can be found in [FFRR15].

5.2. Evaluation Relative to the Requirements

This section presents an evaluation of the prototype and the containerization approach, relative to the assessment criteria from Chapter 3 (the requirements, for short). With the prototype and test results in hand, it is assessed to what extent the requirements are met. This is compared to how the virtual machine-based solution performs relative to those same requirements. The section structure mirrors that of Chapter 3.

Note that great care needs to be taken with respect to what is being compared, between the containerization prototype on the one hand, and the existing virtual machine-based approach on the other. The containerization prototype is meant to serve as a basis for replacing the virtual machine backend now in use at AlpEnDAC, it cannot be directly compared to the full implementation (backend and frontend) now in production. In addition, some of the requirements, like adherence to best Docker practices, are specific to the containerization approach. They simply do not apply to a virtual machine-based alternative. Finally, the existing virtual machine-based approach was not implemented with the requirements from this thesis at hand. In other words, even where this was not in fact done, some of the requirements might in principle be achievable using virtual machines.

5.2.1. Volumes as Data Interfaces

From the point of view of a containerized application, the data interface to a volume is just a well-defined mount point. So long as the relevant input data are present at this location, it is irrelevant how they got there. Output data will be written to the relevant output location irrespective of whether it will be processed further from there. In this sense the interface is very stable, fulfilling requirement R5 (see Table 5.1).

When a volume is mounted to such a well-defined mount point, then these interfaces also become persistent, flexible, and transparent, fulfilling the requirements R2, R3, and R4. The sub-requirement R3.A was respected in the prototype implementation, while the sub-requirements R3.B and R3.C are inherently fulfilled by the properties of Docker volumes. The amount of data that can be handled by a volume is limited by the availability of underlying system resources. That is, disk space on the Docker host for local volumes or the throughput of the underlying network for remote volume drivers. These factors are free to scale with the available infrastructure, meeting requirement R1.

In conclusion, volumes are well suited to the requirements for heterogeneous data interfaces. They fulfill all the stated requirements from Table 5.1.

The production virtual machine-based implementation uses an NFS mount for the input data, and sends the output data to an AlpEnDAC server using HTTP and FTP (S. Hachinger, private communication). The NFS mount could in principal be replaced by anything else that can be mounted into a virtual machine. As with the container the interface here is really a stable mount point.

For the output data the virtual machines use a network service along with the associated interfaces. This approach is less stable than that for the container. If the AlpEnDAC server were to change, then so would this mechanism. It is also less modular and associated with

5. Evaluation and Assessment

increased maintenance. The network service needs to be installed and maintained within the same virtual machine template as the main Flexpart application. Note that using separate virtual machines for every microservice is not practical due to the overhead and uptime associated with them. Each virtual machine would require its own dedicated system resources and operating system kernel.

The output is also not locally persistent beyond the lifetime of individual virtual machines. On the container it is persistent for the life time of the Docker host rather than individual containers.

These considerations are summarized in Table 5.1:

Requirement (from Section 3.1)	Container	Virtual Machine
R1: Handle large amounts of data	yes	yes
R2: Persistent output data	yes	once transferred
R3: General and flexible interface	yes	yes
A: Separate output, input, and configuration interfaces	yes	yes
B: Native support for large range of storage solutions	yes	some
C: Modular extensibility	yes	no
R4: Independent of the data source (transparent)	yes	yes
R5: Stable interface	yes	no

Table 5.1.: Assessment of interface requirements (see Section 3.1) for containers and VMs

5.2.2. Parameterized Containers for Run-Specific Configuration

As indicated by the basic functionality tests (see Section 5.1.1 above), the Flexpart containerization prototype does implement the AlpEnDAC runtime parameters (requirement R1 from Table 5.2). In addition, the general approach of using parameterized containers is easily extensible to support arbitrary parameters (requirement R2 from Table 5.2, see also Section 4.1.4).

Trivially, the existing production implementation also implements the AlpEnDAC runtime parameters. When an AlpEnDAC user enters these parameters in the AlpEnDAC web interface (recall Figure 3.1), an internal data product ID is generated. This data product ID is injected into a newly instantiated virtual machine, just like for the “autolaunch” test machines from Section 5.1. The virtual machine then uses the data product ID to request the full configuration parameters from the server running the web interface. Clearly, the number of parameters that are sent, could be arbitrarily extended.

The container- and virtual machine-based approaches both work for the existing AlpEnDAC use case. If there is an advantage to the container prototype, it’s that it can also be used quite independently outside of that use case.

The approach also fulfills the ease of use requirement, namely, that individual Flexpart runs be configurable via a single configuration file (requirement R6 from Table 5.3 in the following subsection).

Requirement (from Section 3.2)	Container	Virtual Machine
R1: Must implement the AlpEnDAC parameters	yes	yes
R2: Must be easily extensible	yes	yes

Table 5.2.: Assessment of configuration requirements (see Section 3.2) for containers and VMs

5.2.3. Ease of Use Requirements

The number of steps needed to get from a working Docker installation to a successful Flexpart run (see requirement R1 from Table 5.3) are a quantifiable ease of use criterion. These steps are documented (requirement R2 from Table 5.3) within the `README.md` file of the `flexpart_containerization` Git repository (see Appendix A). This documentation is reproduced in Appendix B of this thesis for convenience. The number of steps needed currently stands at eight. They describe a use case with a SSHFS-based volume for input data. Each step, with the exception of “obtaining access to input data”, corresponds to a single console command.

Note that a certain amount of good faith is required in the assessment of this criterion. The number of “steps” could be reduced to one if they were all placed into a single script. However, if the execution of that script would then require constant user intervention to work for individual Flexpart runs or for its use on different systems, nothing would be gained.

Note also that the eight steps described in the `flexpart_containerization` repository should work out of the box on any Linux system with a sufficiently up-to-date version of Docker. They are independent of the distribution and package manager used.

Compare this to the compilation instructions on the Flexpart website², featuring some 36 console commands, divided into nine different steps, a warning that this guide is not up-to-date, and labouring under the assumption that users following this guide happen to have all the correct sources on them (without any version level documentation).

Since the existing virtual machine-based implementation was never intended for independent distribution outside of AlpEnDAC, the setup requirements (R1 and R2) are not applicable to the virtual machine case. It is concluded that the containerized Flexpart prototype offers a significant improvement compared to manual compilation on differing Linux systems.

There is some room for improvement with respect to the ease of obtaining Flexpart input data. Currently this step essentially requires contacting Dr. Hachinger making a request to him, or otherwise obtaining alternate Flexpart input data from elsewhere. It would be beneficial if a streamlined way to providing Flexpart input data for use with this repository were available. This suggestion is left as a recommendation.

The remaining ease of use requirements (R3 to R7 from Table 5.3) all concern the provision of run-time and build-time dependencies, an initial model configuration, as well as documentation of those features. All of these requirements are met by the containerized Flexpart prototype. Moreover, they are met in a way that is inherent to the general containerization approach. As a result, these requirements will be met by any environmental simulation

² Hints for installing FLEXPART v8.2. Retrieved September 1st, 2018, from <https://www.flexpart.eu/wiki/FpInstall>

5. Evaluation and Assessment

containerized using the blueprint developed as part of this work. With the exception of the documentation requirements, these requirements are also met by the virtual machine-based implementation (see Table 5.3).

Requirement (from Section 3.3)	Container	Virtual Machine
R1: Minimal number of setup steps	8	n/a
R2: Documentation of setup steps	yes	n/a
R3: Provide all run-time dependencies	yes	yes
R4: Documentation of the build process	self documenting	no
R5: Provide a full model configuration	yes	yes
R6: Provide a configuration interface	yes	yes
R7: Documentation of bundled components	self documenting	no

Table 5.3.: Assessment of ease of use requirements (see Section 3.3) for containers and VMs

5.2.4. Reusability and Maintenance Requirements

In order to assess the maintenance requirements, a quick description of how routine upgrades are performed within the containerization approach is required (providing this description is requirement R2 from Table 5.4).

Within a given container, the base operating environment, the containerized application itself, any dependencies, and any bundled configuration, may all require upgrades or modification (see also Chapter 4).

Upgrading the base operating environment, is generally as easy as pulling the newest version of the relevant base image from Docker Hub, and rebuilding the container image with it. This process is also the best practice way of upgrading the base operating environment. Generally speaking, the base operating environment should never be upgraded via an additional image layer on top of the base image.

Upgrading dependencies, or anything else installed within the container image via a package manager, is a matter of changing the Dockerfile to install the desired upgraded version of the relevant packages, and then rebuilding the container image. Bundled configuration, or anything else from the build context, can be freely altered before the image is rebuilt.

This flexibility is true also of the source files used to build a particular version of the main containerized application as part of a multi-stage build. If that application is to be upgraded, for example from Flexpart version 9.0 to 10.0, the new sources need to be obtained, and the image needs to be rebuilt. Doing so may well require upgrades to the dependencies and bundled configuration. Checking that everything still works post upgrade should always be part of the upgrade process.

The upgrade workflow, changing the Dockerfile and/or the build context and then invoking `docker build` to rebuild the container image, is generally simpler than the analogous workflow for virtual machines: Upgrading a virtual machine generally involves instantiating the virtual machine from the old template, manually performing any upgrades within the virtual machine, and then creating a new virtual machine template. Instead of a two step process of

altering the build instructions and letting `docker build` do the rest, it is a three step process requiring manual work within the virtual machine. Thus, requirement R1 from Table 5.4 is met.

The containerization approach is inherently modular so long as the rule that each container image should contain exactly one service is respected (requirement R3 from Table 5.4). Finally, the prototype is publicly available on GitHub (see Appendix A and requirement R4).

Requirement (from Section 3.4)	Container	Virtual Machine
R1: Ease of routine maintenance	good	less good
R2: Documentation on routine upgrades	yes	n/a
R3: Modularity of the approach	very	less so
R4: Availability via open-source repository	yes	n/a

Table 5.4.: Assessment of reusability and maintenance requirements (see Section 3.4) for containers and VMs

5.2.5. Reproducibility Requirements

As shown in Section 5.1.2, the reproducibility of simulations was dependent on the level of compiler optimization, but could be achieved with appropriate options. There is no guarantee that this holds true of every system and microarchitecture. Where output data for simulations using the same input data and configuration does diverge, an open source containerization approach at least provides full transparency about how the simulation was built. Requirement R1 from Table 5.5 is thus largely met.

The current Flexpart prototype stores the run-specific configuration file along with the output data to facilitate reproducibility (see requirement R3 from Table 5.5). However, it does not currently record the input data source used, or the exact version of the container within which the simulation was run, leaving some room for improvement on requirement R2. That being said, reproducing a given Flexpart run is as easy as reusing the configuration file from that run (assuming access to the same input data).

To ensure anyone reproducing results can also check the full method used, the entire prototype is available on GitHub (See Appendix A and R4 from Table 5.5). An extensive discussion how Docker and Open Source can tie together to improve both reproducibility and practical adoption of complex software frameworks can be found in [KN17].

Since the existing virtual machine-based implementation is not intended for distribution to different platforms outside the AlpEnDAC infrastructure the reproducibility requirements once again don't apply or else are trivially achieved.

Requirement (from Section 3.5)	Container	Virtual Machine
R1: Identical input should yield identical output where possible	yes	n/a
R2: Information needed for reproduction should be stored with the output	mostly	n/a
R3: Given the configuration and input data, rerunning a simulation should be easy	very easy	n/a
R4: Open-source license and repository	yes	no

Table 5.5.: Assessment of reproducibility requirements (see Section 3.4) for containers and VMs

5.2.6. Remaining Requirements

The prototype is almost entirely composed from free and open source components. Debian prides itself on being free software.³ Docker CE is built from open source components.⁴ The prototype itself has been published publicly on GitHub under the GNU General Public License (version 3).⁵ Best Docker practices have been discussed throughout this work. These points cover the remaining requirements (see Table 5.6).

Requirement (from Sections 3.6 and 3.7)	Container	Virtual Machine
R1: Prefer the use of open-source software	entirely open-source	n/a
R2: Should itself be open-source	yes	n/a
R3: Available via public repository	yes	n/a
R1: Adhere to best Docker practices	yes	n/a

Table 5.6.: Assessment of remaining requirements (see Sections 3.6 and 3.7) for containers and VMs

In conclusion, the requirements laid out in Chapter 3 have largely been met, save for small details as noted in the above sections. While the existing virtual machine-based solution works, the modularity, lack of overhead, ease of use, as well as the self documenting nature of the Docker prototype, offers significant comparative advantages. Especially when considering the long term benefit of greater ease of maintenance, these advantages almost certainly justify the initial investment of making the switch.

³ What Does Free Mean? Retrieved September 1st, 2018, from <https://www.debian.org/intro/free>

⁴ Docker CE on GitHub. Retrieved September 1st, 2018, from <https://github.com/docker/docker-ce>

⁵ flexpart_containerization on GitHub. Retrieved September 1st, 2018, from https://github.com/Quba42/flexpart_containerization

6. Conclusion

The motivation for this thesis lies in the difficulties associated with the use of complex software frameworks for environmental simulations and the resultant high entry hurdles. The goal was to explore the use of containerization (via Docker) as a strategy to reduce these entry hurdles and improve the ease of use, thus facilitating more rapid and well-grounded advances in an area of research that is of great practical importance but also of interest from a computer science perspective.

The method was to construct and use a containerization prototype for a specific example of a simulation embedded in the AlpEnDAC and GeRDI projects, all while seeking to extrapolate a general blueprint for the containerization of environmental simulations from that prototype. The specific example used was the Flexpart particle dispersion model, a version of which was already implemented at AlpEnDAC for the provision of on-demand simulations using virtual machines.

First a set of assessment criteria, applicable to both the specific prototype and the general approach, was developed in Chapter 3. These criteria include requirements for various data interfaces, reusability and maintenance, reproducibility, and openness (in the sense of free and open source software), which contribute to lowering entry hurdles and achieving other properties that are desirable in such a framework.

Once the criteria were clearly laid out, the implementation of a containerized Flexpart prototype for AlpEnDAC was undertaken. As a matter of clean design, as well as the basis for a general blueprint, the problem of containerizing the application was subdivided into several distinct subproblems, as follows:

1. Providing the base operating environment.
2. Providing the application itself.
3. Providing runtime dependencies.
4. Providing bundled configuration.
 - a) Providing a fully working default configuration.
 - b) Identifying parameters that need to be modifiable for individual runs.
 - c) Providing a configuration interface for the identified parameters.
5. Providing data interfaces.

This general taxonomy of the containerization process is itself an important result of this work.

6. Conclusion

Key techniques for solving these subproblems, employed in this thesis, include the use of a multi-stage Docker build for compiling code from source; the provision of parameterized containers for the configuration interface; and volume mounts as the principle data interfaces.

In addition to the container providing Flexpart simulations, the prototype includes a separate container providing QuickLook visualizations for Flexpart output data. This additional container serves as a proof of concept for how different containers can interact with one another, as well as a concrete implementation of a post-processing step needed at AlpEnDAC.

In Section 4.2 a general architecture for the containerization of environmental simulations, including the relevant data workflow, was provided. This architecture, visualized in Figure 4.2, is another central result of this thesis. Possible modifications and extensions to the architecture were discussed, for example, the possibility of extending the architecture with additional interfaces, or replacing the central runtime container with a number of parallelized containers all working on the same data volumes.

In Chapter 5 the prototype as well as the general approach were evaluated against the given criteria, and compared on this basis to the current virtual machine-based approach. The prototype was successfully tested both on and off the AlpEnDAC infrastructure (the LRZ “compute cloud”).

As part of these tests, the virtual machine uptime overhead of the current implementation was measured at 80 ± 10 s. This is the time it takes for virtual machines to be instantiated and boot, from when their instantiation is first requested by a user action, to the point at which they first start processing user input. This time is identified as overhead that could be feasibly eliminated using the container approach, provided a container infrastructure featuring at least one permanent Docker host is used.

It was established that both the concrete prototype, as well as the containerization approach in general, satisfy the criteria. Furthermore, containerization offers significant advantages over a virtual machine-based approach, with respect to several criteria including ease of use, reusability and maintenance, and reproducibility. The increased modularity, the self documenting and automated aspect of the Docker build process, should result in a decrease in the amount of effort required for routine maintenance. Collectively these advantages almost certainly justify the relatively modest up-front effort of containerizing environmental simulations.

Outlook/Future Work Since this thesis has sought to provide a blueprint for the containerization of environmental simulations by proceeding from the Flexpart example, the next logical step would be to put the generality of the approach to the test, by containerizing additional simulations. Hysplit, for example, is also in use at AlpEnDAC; currently displays similar entry hurdles; and may make a good choice.

The prototype has been made available as open source project on GitHub under the terms of the GNU General Public License (version 3). This could be incorporated with the upstream Flexpart project in the spirit of giving back to the Flexpart community. With some adjustments to the repository away from the needs of this thesis, a corresponding outreach should be attempted.

Eventually one might move from the approach of building individual containerized applications on individual Docker hosts, to the provision of a full-fledged container infrastructure, using a container-orchestration platform like Kubernetes. With such a platform, the potential for containerized environmental simulations, to be distributed over multiple containers in a parallelized, scalable fashion, might be explored.

A. List of Git Repositories

This appendix provides a list of Git repositories relevant to this work:

- The main repository of this thesis is located on the LRZ GitLab service. It contains this work's entire version history including incomplete drafts, raw data, literature, the LaTeX sources, and all image sources for any Docker containers developed (as submodules). As a result this Git repository is kept private and requires both a valid account and the permission of the author to be accessed. Given access, the repository can be found at:
https://gitlab.lrz.de/ru34wab/bachelor_thesis.
- There is also a public Git repository containing the containerized QuickLook and Flexpart prototypes developed as part of this work:
https://github.com/Quba42/flexpart_containerization
- The original QuickLook sources can be found here:
<https://bitbucket.org/radekhofman/quicklook>
- Information on the Flexpart development Git repository can be found here:
<https://www.flexpart.eu/wiki/GitFlexpart>
- The Debian Stretch source package for Flexpart, along with a clone link for its Git repository, can be found here:
<https://packages.debian.org/source/stretch/flexpart>
- The SSHFS volume driver for Docker can be found here:
<https://github.com/vieux/docker-volume-sshfs>
- A volume driver supporting NFS (amongst others) can be found here:
<https://github.com/ContainX/docker-volume-netshare>

All links were accessed on August 26th, 2018.

With the tag `final-thesis` at the commit “Merge branch ‘final_touche’s” (with the hash `e8aa8eb60fcf533d753d2febcbddc9d8da14dc1a`) in the `bachelor_thesis` repository, this thesis has reached its current state (just prior to the addition of this notice).

This includes a submodule reference to the commit “Merge branch ‘add_license’” (with the hash `3281388b44157eca12d8dc76fbee84583b290c15`) in the `flexpart_containerization` repository.

B. Flexpart Container Quick Start Guide

This appendix provides a quick start guide (documentation) on how to get to a first Flexpart simulation, given a working Docker host and access to the `Quba42/flexpart_containerization` Git repository (https://github.com/Quba42/flexpart_containerization).

The same information is also contained in the repository's `README.md` file.

Note that these instructions presume a Linux system with a working Docker installation. For more on obtaining Docker, see <https://docs.docker.com/install/>. Ultimately, Docker is likely to be available via your distributions package manager.

Note also that the `Quba42/flexpart_containerization` repository was developed and tested with Docker version 18.03.1. It may not work with much older versions of Docker.

Steps to get to your first Flexpart simulation:

1. Clone the `Quba42/flexpart_containerization` repository to any Linux host with a working Docker installation:

```
git clone https://github.com/Quba42/flexpart_containerization.git
```
2. Change to the repositories base directory:

```
cd <path_to_git_repo>/flexpart_containerization/
```
3. Build the main container image:

```
./build_alpendac_prototype.sh
```
4. Obtain access to Flexpart input data. (Currently this involves contacting someone at AlpEnDAC about getting access).
5. Install `vieux/sshfs` (<https://github.com/vieux/docker-volume-sshfs>) in case you want to use the `sshfs` based data repository for input data:

```
docker plugin install vieux/sshfs
```
6. Create the input data volume:

```
docker volume create -d vieux/sshfs -o \
sshcmd=weatherdata@<alpendac_sshfs_repo_ip_addr>:/weatherdata/GFS \
-o password=<password> -o port=<port> flexpart_input_sshfs
```
7. Create a local output data volume:

```
docker volume create flexpart_output_local
```
8. Run your first example simulation using default parameters:

```
./run_alpendac_prototype.sh
```

Note that the creation of any volume containing relevant Flexpart input data will work instead of steps 4-6 above. All links were accessed on September 1st, 2018.

C. Table of User Parameter Tests

DIRECTION	START_DATE	START_HOUR	RUN_LENGTH	LATITUDE	LONGITUDE
-1	2016-09-19	00	3	47.5	11
-1	2016-09-19	00	2	47.5	11
-1	2016-09-19	09	2	47.5	11
-1	2016-09-19	07	2	47.5	11
1	2016-09-19	03	2	47.5	11
1	2016-09-19	03	2	20	120
-1	2016-09-19	03	2	20	120
-1	2018-06-03	23	4	45.67	12.34

List of Figures

2.1.	Google Trends for <i>Docker</i> and <i>Operating-system-level virtualization</i>	3
2.2.	Container vs. virtual machine comparison	4
2.3.	Container and image layers (based on the ubuntu:15.04 image)	5
2.4.	Docker platform architecture	6
2.5.	The GeRDI Self-Contained Systems Architecture. [dSHWK18]	14
3.1.	The AlpEnDAC web interface for the configuration of Flexpart runs.	18
4.1.	Quicklook container visualization example using default configuration	32
4.2.	An architecture for the containerization of environmental simulations	33
5.1.	The Open Nebula web interface with test machines	35
5.2.	QuickLook visualization of the march= native reproducibility test	37

List of Tables

2.1. Summary of runtime and launch-time data interface mechanisms	10
3.1. Runtime configuration parameters for Flexpart runs in AlpEnDAC.	19
5.1. Assessment of interface requirements for containers and VMs	40
5.2. Assessment of configuration requirements for containers and VMs	41
5.3. Assessment of ease of use requirements for containers and VMs	42
5.4. Assessment of reusability and maintenance requirements for containers and VMs	43
5.5. Assessment of reproducibility requirements for containers and VMs	44
5.6. Assessment of remaining requirements for containers and VMs	44

List of Listings

4.1. Compiling Flexpart	25
4.2. The Flexpart multi-stage build	27
4.3. The Flexpart model configuration and parameterization	28
4.4. The QuickLook runtime dependencies in the Dockerfile	31

Bibliography

- [Ada06] Michael D. Adams. JasPer software reference manual (version 1.900.0). *ISO/IEC JTC*, 1:37, 2006.
- [AK00] M. D. Adams and F. Kossentini. JasPer: a software-based JPEG-2000 codec implementation. In *Proceedings 2000 International Conference on Image Processing (Cat. No.00CH37101)*, volume 2, pages 53–56 vol.2, 2000.
- [BAS⁺13] J. Brioude, D. Arnold, A. Stohl, M. Cassiani, D. Morton, P. Seibert, W. Angevine, S. Evan, A. Dingwell, J. D. Fast, R. C. Easter, I. Pisso, J. Burkhardt, and G. Wotawa. The lagrangian particle dispersion model FLEXPART-WRF version 3.1. *Geoscientific Model Development*, 6(6):1889–1904, 2013.
- [BR17] M Ali Babar and Ben Ramsey. *Evaluating Docker for Secure and Scalable Private Cloud with Container Technologies*. CREST, University of Adelaide, Adelaide, Australia, 2017.
- [dSHWK18] Nelson Tavares de Sousa, Wilhelm Hasselbring, Tobias Weber, and Dieter Kranzlmüller. Designing a generic research data infrastructure architecture with continuous software engineering. In *Software Engineering (Workshops)*, pages 85–88. CEUR-WS. org, 2018.
- [FFRR15] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and Linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.
- [GAB⁺17] Richard Grunzke, Tobias Adolph, Christoph Biardzki, Arndt Bode, Timo Borst, Hans-Joachim Bungartz, Anja Busch, Anton Frank, Christian Grimm, Wilhelm Hasselbring, et al. Challenges in creating a sustainable generic research data infrastructure. *Softwaretechnik-Trends*, 37(2):74–77, 2017.
- [GBRM⁺09] Jesus M Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.
- [HHM⁺16] S. Hachinger, C. Harsch, J. Meyer-Arneke, A. Frank, H. Heller, and E. Giemsa. On-demand simulation of atmospheric transport processes on the AlpEnDAC cloud. *AGU Fall Meeting Abstracts*, pages IN21B–1735, 2016.
- [Hog14] Scott Hogg. Software containers: Used more frequently than most realize. *Network World, Inc*, 2014. <https://www.networkworld.com/article/2226996/cisco->

- subnet/software-containers-used-more-frequently-than-most-realize.html – accessed on 1. September 2018.
- [Kha16] Latika Kharb. Automated deployment of software containers using dockers. *International Journal of Emerging Technologies in Engineering Research (IJETER)*, 4(10):1–3, 2016.
- [KN17] Christian Knoth and Daniel Nüst. Reproducibility and practical adoption of GEOBIA with open-source software in Docker containers. *Remote Sensing*, 9(3):290, 2017.
- [Kol06] Kirill Kolyshkin. Virtualization in Linux. *White paper, OpenVZ*, 3:39, 2006.
- [MK79] MD McIlroy and BW Kernighan. *Unix Programmer’s Manual – Volume 1*. Murray Hill, New Jersey, seventh edition, 1979.
- [MK12] D Morton and M Krysta. Analysis of FLEXPART performance in an operational atmospheric transport modeling environment at CTBTO. In *EGU General Assembly Conference Abstracts*, volume 14, page 6034, 2012.
- [MNV⁺17] Barend Mons, Cameron Neylon, Jan Velterop, Michel Dumontier, Luiz Olavo Bonino da Silva Santos, and Mark D Wilkinson. Cloudy, increasingly FAIR; revisiting the FAIR data guiding principles for the European Open Science Cloud. *Information Services & Use*, 37(1):49–56, 2017.
- [PSG⁺17] I. Pisso, E. Sollum, H. Grythe, N. Kristiansen, M. Cassiani, S. Eckhardt, R. Thompson, C. Groot Zwaaftnik, N. Evangeliou, T. Hamburger, H. Sodemann, L. Haimberger, S. Henne, D. Brunner, J. Burkhart, A. Fouilloux, X. Fang, A. Phillip, P. Seibert, and A. Stohl. The lagrangian particle dispersion model FLEXPART version 10. In *EGU General Assembly Conference Abstracts*, volume 19, page 19540, 2017.
- [SFF⁺05] A. Stohl, C. Forster, A. Frank, P. Seibert, and G. Wotawa. Technical note: The lagrangian particle dispersion model FLEXPART version 6.2. *Atmospheric Chemistry and Physics*, 5:2461–2474, 2005.
- [SWSKK95] Andreas Stohl, Gerhard Wotawa, Peter Seibert, and Helga Kromp-Kolb. Interpolation errors in wind fields as a function of spatial and temporal resolution and their impact on different types of kinematic trajectories. *Journal of Applied Meteorology and Climatology*, 34:2149–2165, 1995.
- [Wal17] Stephen Walli. Demystifying the Open Container Initiative (OCI) specifications, 2017. <https://blog.docker.com/2017/07/demystifying-open-container-initiative-oci-specifications/> – accessed on 1. September 2018.
- [WDA⁺16] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The FAIR guiding principles for scientific data management and stewardship. *Scientific data*, 3, 2016.
- [WHNW18] Jens Weismueller, Stephan Hachinger, Hai Nguyen, and Tobias Weber. Addressing knowledge and know-how biases in the environmental sciences with modern

data and compute services. In *EGU General Assembly Conference Abstracts*, volume 20, page 4399, Vienna, Austria, 2018.

- [Wor18] World Meteorological Organization. *Manual on Codes (WMO-No. 306) – FM 92 GRIB edition 2*, 2018. http://www.wmo.int/pages/prog/www/WMOCodes/WMO306_vI2/LatestVERSION/LatestVERSION.html – accessed on 1. September 2018.

Acknowledgements

I would like to thank Tobias Weber and Dr. Stephan Hachinger for supervising this work and for always providing the right mixture of patience and encouragement whenever my progress was lacking. I want to thank the Munich Network Management (MNM) Team and Prof. Dr. Dieter Kranzlmüller for providing me with the opportunity to work on this topic.

I want to thank my parents for supporting me throughout my education as well as for proofreading drafts at short notice. I want to thank Regina Sachsenhauser for her love and support, more proofreading, and a particularly insightful suggestion for flipping the orientation of Figure 4.2.

I want to thank my friend Kevin Edmonds for helping me with several thorny LaTeX problems. I want to thank the rest of my gang for putting up with me working through most of our long planned joint holiday. Finally, I want to thank, and apologize to anyone else I have forgotten.